AD-A262 109

Report No. NAWCADWAR-92100-50

DTIC
ELECTE
MAR 11 1993
S C D

MACRO FUNCTION LANGUAGE HARDWARE MANUAL

Timothy Monaghan and Robert Peck
Mission Avionics Technology Department (Code 5051)
NAVAL AIR WARFARE CENTER
AIRCRAFT DIVISION WARMINSTER
P.O. Box 5152
Warminster, PA 18974-0591

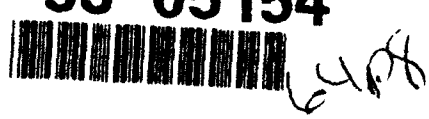
OCTOBER 1992

FINAL REPORT

Approved for Public Release; Distribution is Unlimited.

Prepared for
Mission Avionics Technology Department
NAVAL AIR WARFARE CENTER
AIRCRAFT DIVISION WARMINSTER
P.O. Box 5152
Warminster, PA 18974-0591

93-05154



93 3 10 035

NOTICES

REPORT NUMBERING SYSTEM — The numbering of technical project reports issued by the Naval Air Warfare Center, Aircraft Division, Warminster is arranged for specific identification purposes. Each number consists of the Center acronym, the calendar year in which the number was assigned, the sequence number of the report within the specific calendar year, and the official 2-digit correspondence code of the Functional Department responsible for the report. For example: Report No. NAWCADWAR-92001-60 indicates the first Center report for the year 1992 and prepared by the Air Vehicle and Crew Systems Technology Department. The numerical codes are as follows:

CODE	OFFICE OR DEPARTMENT
00	Commanding Officer, NAWCADWAR
01	Technical Director, NAWCADWAR
05	Computer Department
10	AntiSubmarine Warfare Systems Department
20	Tactical Air Systems Department
30	Warfare Systems Analysis Department
50	Mission Avionics Technology Department
60	Air Vehicle & Crew Systems Technology Department
70	Systems & Software Technology Department
80	Engineering Support Group
90	Test & Evaluation Group

PRODUCT ENDORSEMENT — The discussion or instructions concerning commercial products herein do not constitute an endorsement by the Government nor do they convey or imply the license or right to use such products.

Reviewed By: Laurence R. Hall **Date:** 1/4/93
Branch Head

Reviewed By: [signature] **Date:** 12/3/92
Division Head

Reviewed By: [signature] **Date:** 12/14/92
Director/Deputy Director

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1992	Final

4. TITLE AND SUBTITLE
MACRO FUNCTION LANGUAGE HARDWARE MANUAL

5. FUNDING NUMBERS
W044660321N

6. AUTHOR(S)
Timothy Monaghan and Robert Peck

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Mission Avionics Technology Department (Code 5051)
NAVAL AIR WARFARE CENTER-AIRCRAFT DIVISION WARMINSTER
P.O. Box 5152
Warminster, PA 18974-0591

8. PERFORMING ORGANIZATION REPORT NUMBER
NAWCADWAR-92100-50

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Mission Avionics Technology Department
NAVAL AIR WARFARE CENTER-AIRCRAFT DIVISION WARMINSTER
P.O. Box 5152
Warminster, PA 18974-0591

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release; Distribution is Unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report examines the hardware interface required for an efficient implementation of the Macro Function Language (MFL). Traditionally, computer systems were developed by designing hardware, then writing microcode, assembly language, and on up to a common programming language such as ADA or Pascal. This emphasis on hardware was due to the relative high cost of hardware. In the current state of the art, software costs are far greater than hardware over the life of a signal processor. Because of this, the MFL attempts to standardize the software interface to hardware. This interface will standardize the software between signal processors designed to run MFL down to the microcode level.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Macro Function Language (MFL), Signal Processing, Primitives, Multisensor Standard Macro Study	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS RPT.

TABLE OF CONTENTS

DTIC QUALITY INSPECTED 1

Accesion For
NTIS CRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

INTRODUCTION

The Macro Function Language, MFL, is a signal processing language developed out of NADC's MULTISENSOR STANDARD MACRO FUNCTION STUDY contract N62269-79-C-0116 performed by the Submarine Signal Division of Raytheon Company. This study identified and characterized signal processing macro functions which are common to multiple sensor areas. Signal processing macro functions are a complete set of primitive functions, control operators and array transformations for signal processing operations. (See figure 1.) A complete set of primitives provides all the functions needed to solve signal processing problems, and permits the solution to be stated in a form familiar to a signal processing analyst.

NADC formalized this Macro Function Set into the MFL language under contract N62269-83-C-0441 performed by Raytheon. The development of MFL has been an effort to first look at signal processing applications, then at a primitive macro function set that would conveniently express those applications, and finally formalize those primitives into a signal processing language, MFL. This report will take the development process a step further by defining the hardware that will most efficiently execute MFL.

Traditionally, computer systems were developed by designing hardware, then writing microcode, assembly language, on up to a

common programming language such as ADA or Pascal. This emphasis on hardware was due to the relative high cost of hardware. In the current state of the art, software costs are far greater than hardware over the life of a signal processor. Because of this , MFL attempts to standardize the software interface to hardware. This interface will standardize the software between signal processors designed to run MFL down to the microcode level. This report will examine this standard interface and its effect on hardware designs.

This report deals with the hardware considerations that MFL requires for an efficient implementation with only a brief introduction to the language. For a complete description of the language see NADC report #N62269-83-C-0441.

MFL Code

Minimum Function Box Code Primitives

+	Add	∧	And
-	Minus	~∧	Nand
X	Multiply	∨	Or
=	Equal To	~∨	Nor
*	Conjugate Multiply	≤	Greator Than or Equal To
<	Less Than	≥	Less Than or Equal To
>	Greator Than		
⌜	Maximum		
⌞	Minimum		

Minum Function Box Control Operators

A ←— f B	Single Stream
A←—B f C	Corresponding Elements
A←——f /C	Reduction
B f2"f1 C	f1 Operates on Real Part of B & C f2 Operates on Imaginary Part of B & C
'f' C	Combine Function (= RE C f IM C)
A f2 : f1 B	Composite - Executed Left to Right

Array Transformation Code

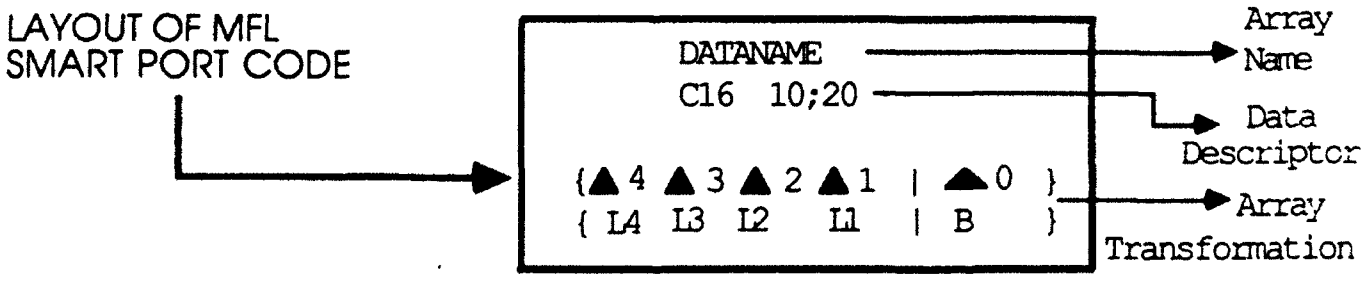


Figure 1

AN MFL OVERVIEW

Signal processing data is highly structured. The data is naturally in the form of arrays of vectors and matrices. This is in contrast to data processing where the data is random and unstructured and the emphasis is on procedural control. The structure on the data must be provided by the language. Exactly as the math of signal processing applies functions to arrays MFL applies functions to arrays. This close tie of MFL to signal processing math makes the language easy to learn for someone who is already familiar with the mathematics of signal processing (in fact too much procedural programming experience can actually be a hindrance to learning MFL). MFL performs array manipulation using a construct called the array transformation. The array transformation, to be discussed later, can conveniently express any sequencing or rearrangement of data within arrays. (eg. -- transpose)

In an efficient vector processor, operations to align and select elements from arrays must occur in parallel with arithmetic operations. These tasks can complicate arithmetic hardware, and the programming of that hardware, if the alignment or shifting must be done in the arithmetic unit. MFL breaks data manipulation and math processing into a natural division of tasks. The data manipulation is done by the MFL array transformation code which gives the programmer a flexible addressing mechanism. The math processing is done by the MFL math primitives.

These two types of MFL code are partitioned graphically as shown in figure 1.

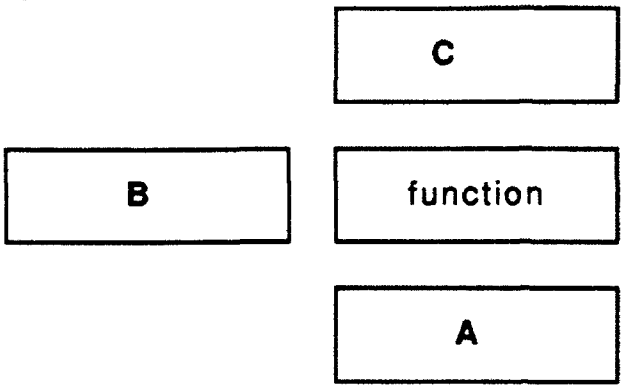
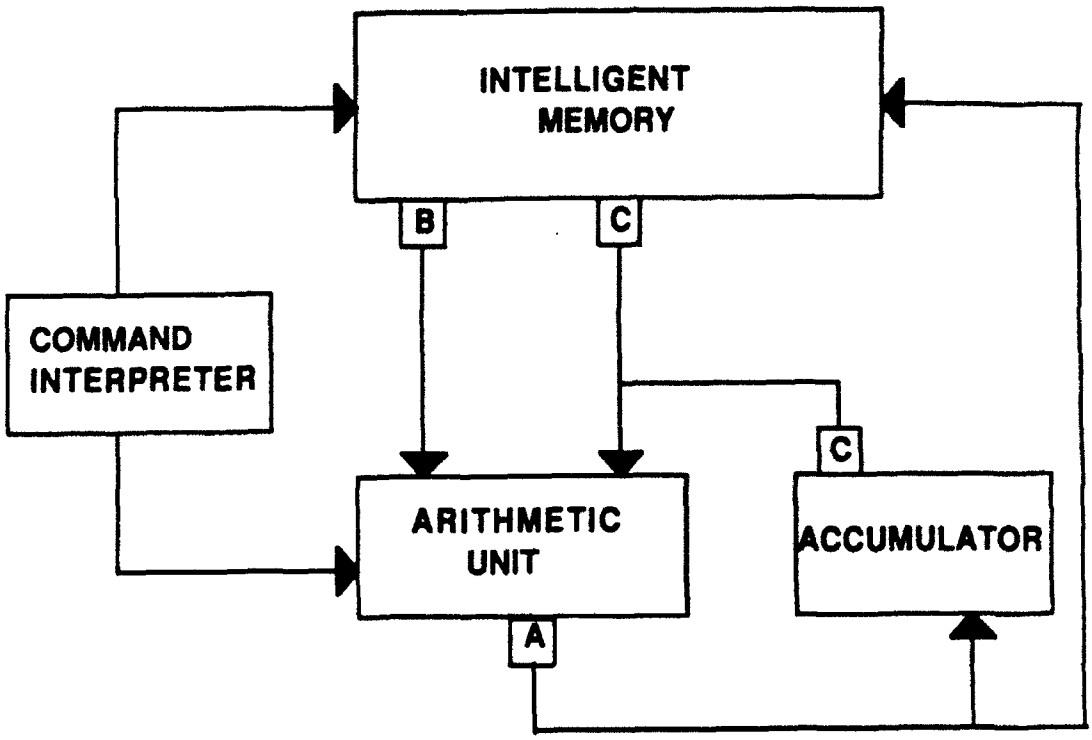


Figure 2

The four boxes in figure 2 represent the four fields of MFL code. The boxes marked A,B, C hold the array transformation code, the code that selects and aligns elements from data arrays in memory, and the data descriptors. The box marked "function" holds the math primitives and control operators. In figure 3 a generic MFL processor is shown. The code from boxes A,B C in figure 2 drive the smart port hardware ports A, B, C. The math primitives, the code in the box marked "function" in figure 2, drive the arithmetic unit in figure 3.



A GENERIC MFL MACHINE

Figure 3

The general format of an array transformation and data descriptor is:

```
        DATANAME
        C16   10;20
{ Δ4  Δ3  Δ2  Δ1 | Δ0 }
{ L4  L3  L2  L1 | B  }
```

The data name, DATANAME is the name of the array. The second line says 16 bit complex data and a 10 by 20 array. The third line, delta 1, 2, 3 and 4, gives the displacement directions, the displacement direction is either in the row, column or depth directions, and delta 0 is the starting point for the read. The fourth line gives the length of each of the displacements, L1, L2, L3, and L4, that is how far the read in that direction is to be, and the boundary mode, B, (wrap around or zero fill,) which tells what to do when the end of the row or column is reached.

For this report it is important to know the graphical layout of MFL code, i.e. the fields that hold the array transformations and data descriptors and the field that holds the mathematical instructions of the signal processing operation. This division buys the programmer a straight forward way of dividing up the signal processing operation and as this report will show, if the hardware is built for the language, it eliminates the need to change microcode with each new application. The microcoding then is part of the hardware design cycle and after that it is not dealt with.

MFL can be viewed as a group of reduced instruction sets optimized for signal processing. Each set can be efficiently implemented directly in hardware to eliminate the need for microcode.

A PROGRAMMING EXAMPLE FROM THE MFL WORKSTATION

A complete MFL workstation, for writing and debugging MFL code, has been implemented on a Macintosh computer. Figure 4 shows a MFL program on the workstation. The program is called dcp_dcr.mfl, and is an adaptive filter for estimating and removing the DC component from a waveform to form a zero-mean waveform. The figure shows the workstation's user interface. A precomputed estimate of DC average, SO, is first subtracted from the input sine wave, X. Then the new DC average is found. Since the sine wave, X, is a vector, it's average is simply the sum of it's elements divided by it's number of elements (denoted by X[l]). This average is weighted by the factor A and used to update the DC average estimate, SO. The figure shows the output, Y, of the program after the program has executed. Data in any of the windows can easily be changed and the algorithm re-executed. This example is given to show the windowed interactive enviroment of the workstation and a complete MFL program.

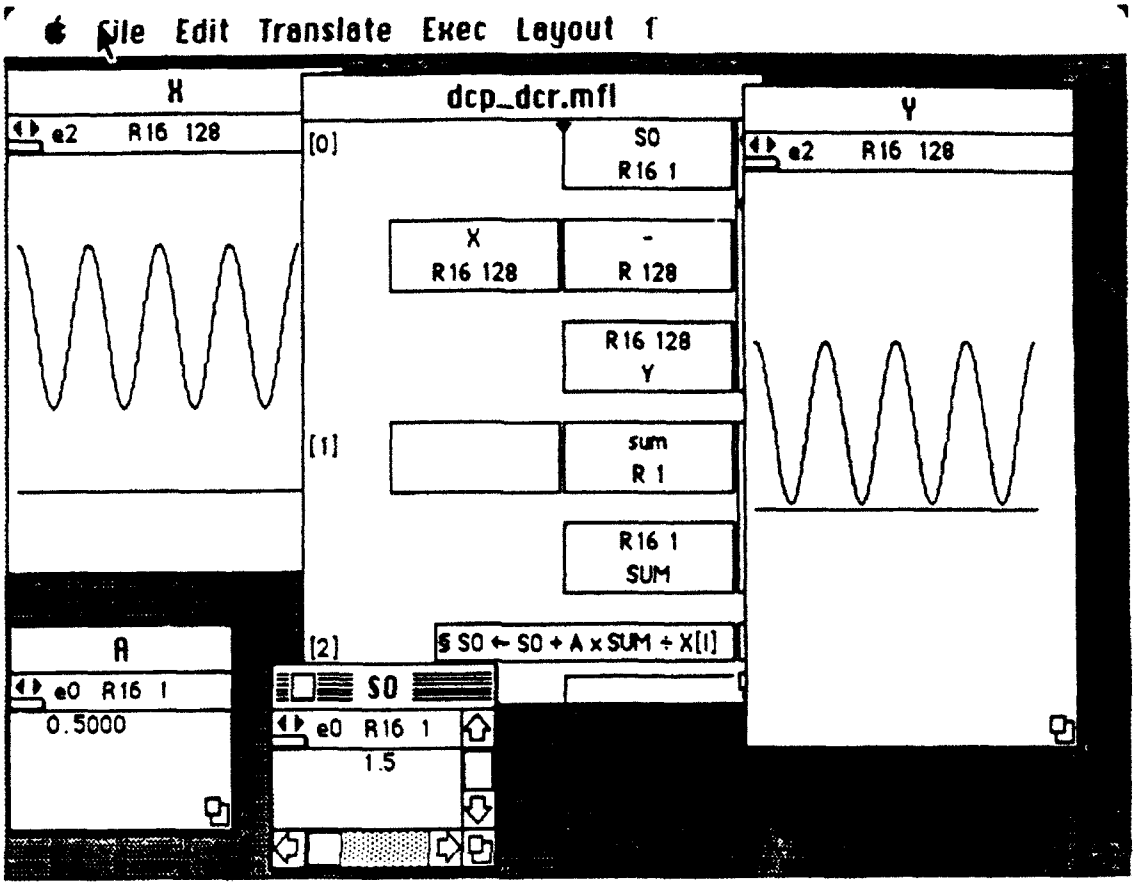


Figure 4 Adaptive DC Removal Program

Basic MFL Design

The MFL language specifications should be used to design an efficient MFL processor. Two key points in MFL processor design philosophy are: (1) an MFL processor is in effect a high level machine, the processor is built to optimize the constructs of the language, and the Assembly language programming for this machine- MFL- is then a high level language; (2) the MFL processor can be viewed as 4 processors running in lock step. The code that drives each of the 4 processors (3 smart ports and the arithmetic unit) is the code in the 4 boxes in the MFL code fields on the left of figure 5. The four MFL processors that the code in each of these fields drive are on the right of figure 5. These four processors can be completely independent at run time or they can be controlled by a Command interpreter (see Command interpreter section - the independence of the four processors can vary).

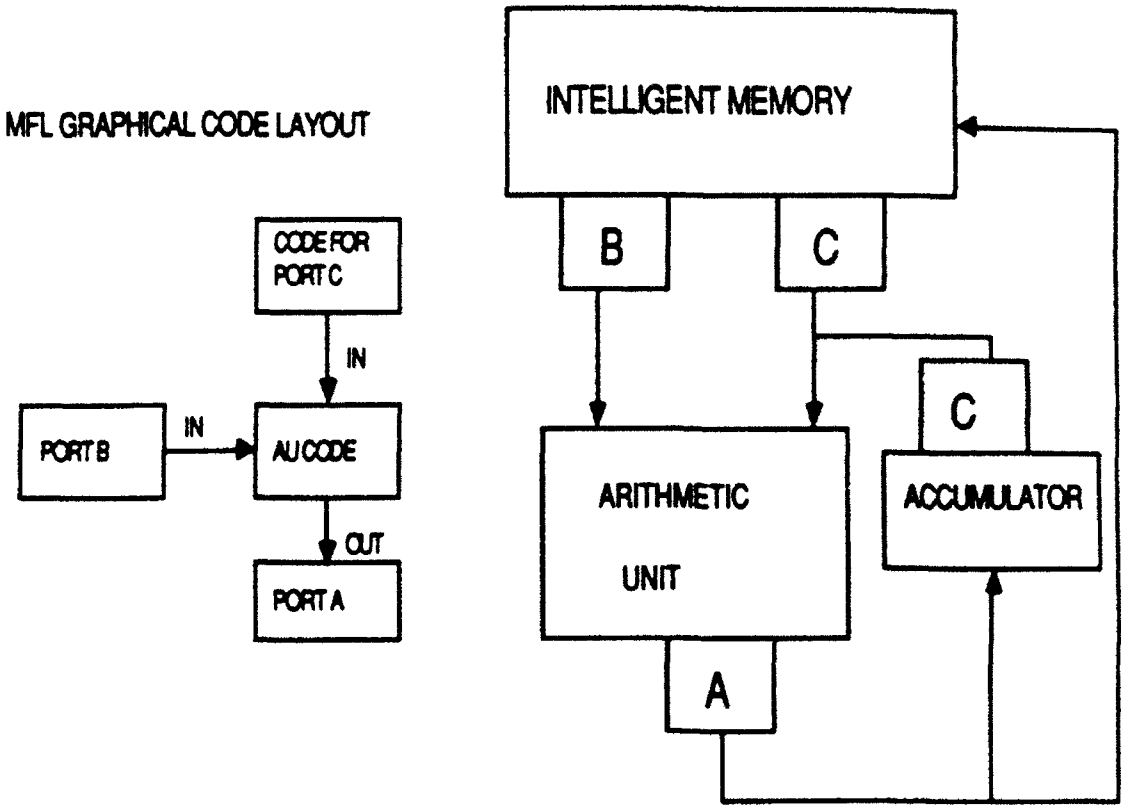


Figure 5

A MFL processor is a three address machine, the three addresses being supplied by smart ports A, B, and C with a Arithmetic Unit optimized to perform the class of algorithms the processor will work on. During each cycle of a MFL processor the two input data elements are supplied to the upper stage of the AU by smart ports B and C and smart port A writes the output of the lower stage into memory. These 4 elements, the Smart Ports and the AU, are the pipe in a MFL processor.

ARITHMETIC UNIT

The MFL Arithmetic Unit is a simplified AU. It doesn't need the shifters, status registers, pre- and post scalers, or rounders of current arithmetic units. No data formatting or sequencing is done in the arithmetic unit-it is strictly a number cruncher. A separate and flexible addressing, sequencing and data formatting mechanism is found in each of the Smart Ports of the MFL processor. When data reaches the AU it is formatted and ready for processing. Therefore, a uniform number representation for the inputs and outputs of the AU is needed and this function is handled by the Data Formatter portion of the Smart Port.

A basic two stage MFL AU is shown in figure 6. This two stage AU, with a ALU and multiplier in the upper stage and a ALU and accumulator in the lower, fits a multiply-accumulate structure of many DSP algorithms. There are many AU structures that fit in a MFL design. A MFL AU is solely the number cruncher of the processor and it should be designed to optimize the class of problems that the processor will work on.

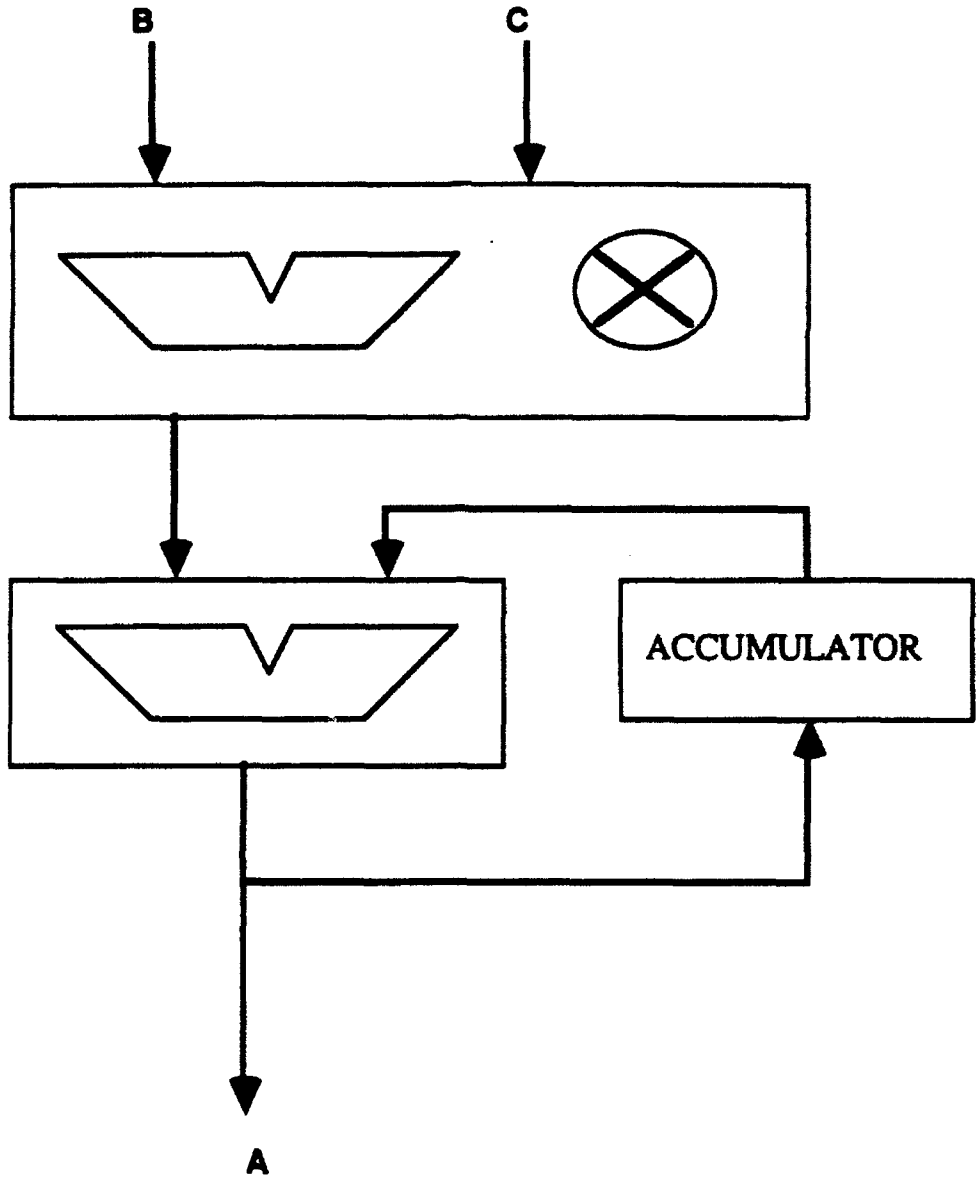


Figure 6

Figure 7 shows the mapping of the AU code of a multiply-accumulate example to the AU shown of figure 6. The MFL code reads, from left to right: multiply the inputs from smart ports B and C then add them to the previous product. This multiply accumulate

form has the structure: (#)(#) + (#)(#) + ... The machine code of the adaptor would select the multiply operation for the upper stage and the add-accumulate operation for the lower stage.

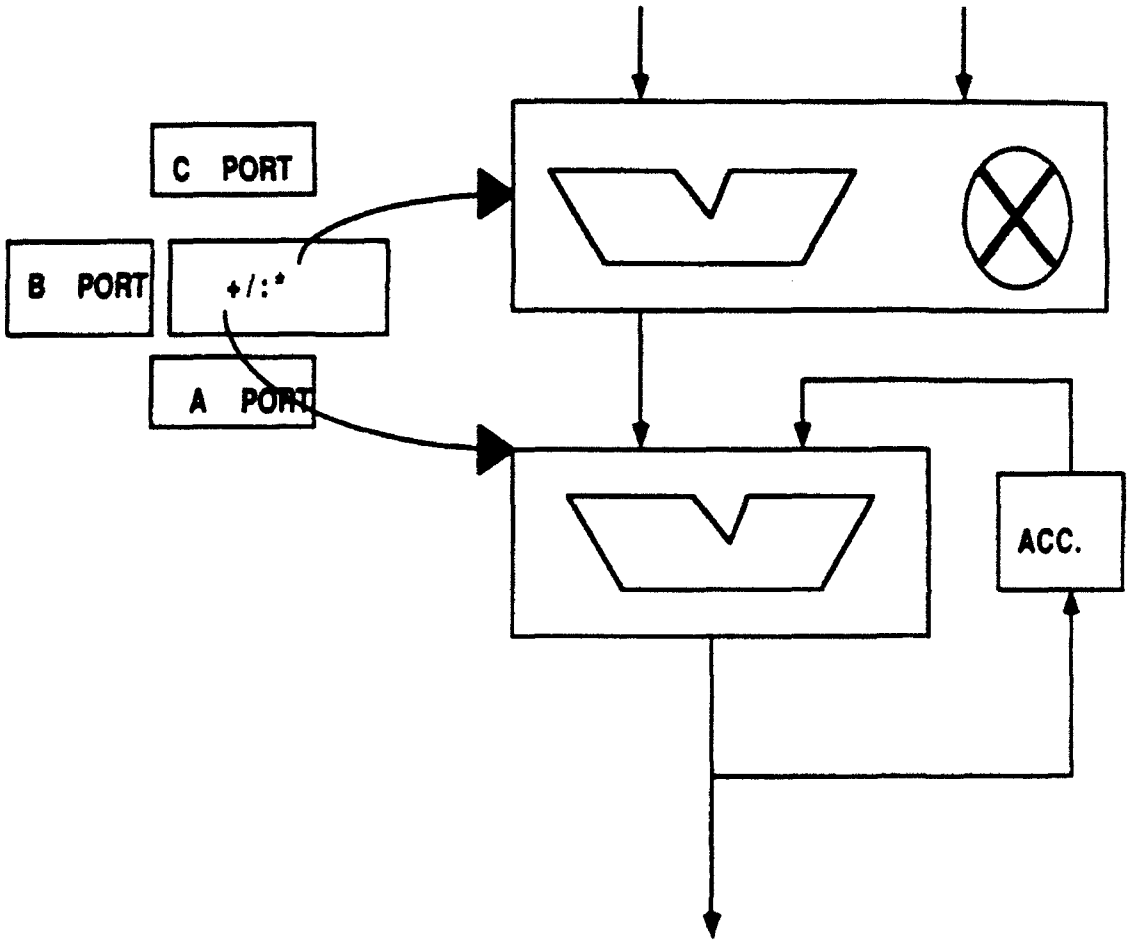


Figure 7

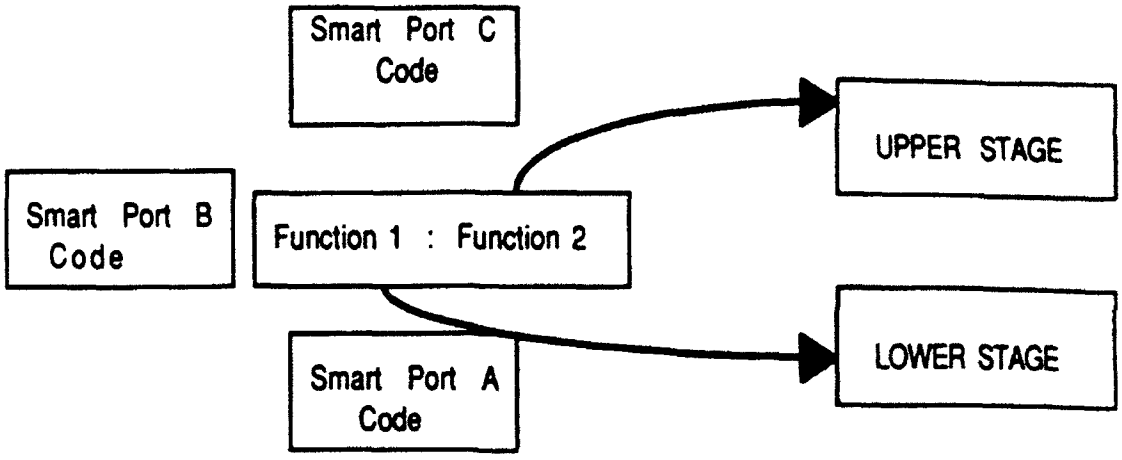


Figure 8

The MFL adaptor, a simple compiler, translates each half of the MFL function field into the machine code that drives the two stages of the MFL AU (figure 8). The left side of the function field drives the lower stage and the right side drives the lower stage. A single stage, triple or even a quadruple stage AU is possible, it merely complicates the adaptor's mapping of the MFL function code to the AU.

SMART PORTS

The MFL Smart Ports are as important as the AU in a MFL processor. A successfully designed MFL processor will be based on the effective implementation of this key element. In this section the Smart Port will be discussed and the basic Smart Port architecture will be covered in detail with particular attention paid to the address generator which is the central feature of the MFL Smart Port processor.

In traditional AU design, the AU must transform data items into a form appropriate for the AU operation. The AU's then switch between modes of data formatting and arithmetic manipulation for each data item. This switching usually requires additional control from the processor control unit. The Smart Port removes this overhead from the AU and the Command Interpreter-- the processor control unit of MFL.

The Smart Port fetches individual elements from memory and delivers them to the AU in the right form and the right order. The Smart Port primitives, the MFL array transformation code, are a set of data manipulation operators. These primitives are translated into the machine code that drives the Smart Port. Once initialized the accessing of data items goes on concurrently with the AU processing. Thus, the Smart Port is a completely separate processor with it's own programming code, the array transformation, that works concurrently and in lock step with the AU. The Smart Port can access long strings of data items for processing by the AU without intervention by the processor control unit. Before taking a detailed look at the Smart Ports address generator architecture it will be

helpful to briefly review the MFL array transformation code since the architecture is designed around this part of the language.

Survey of Array Transformations

(for complete documentation, see NADC Report #N62269-83-C-0441 MACRO FUNCTION SET FORMALIZATION)

The general form of an array transformation is:

```
          DATANAME
          C16   10;20
{ ▲4  ▲3  ▲2  ▲1 | ▲0 }
{ L4  L3  L2  L1 | B  }
```

Where DATANAME is the filename. The second line reads complex 16 bit data in a 10X20 array. The deltas in the third line are the displacements or the directions that the reads take place in. The delta-zero being the starting point of the first read. For example a normal read for a matrix would look like this:

```
DATANAME
C16   10;20
{ j i | 0 }
{ J I | w }
```

delta-1 is an "i" therefore the direction of the first read would be in the row direction. The "L1" below the delta-1, in this example a "I" means read to the end of the row. If the programmer only wanted a certain number of data items from the row read, say 5, the code would look like:

```
DATANAME
C16   10;20
{ j i | 0 }
{ J 5 | w }
```

One could have read the data in transposed order by reading in the column direction for the row elements. The code would the be:

```
DATANAME
C16   10;20
{ i j | 0 }
{ I J | w }
```

This array would then be delivered to the Arithmetic Unit column by column instead of row by row. The delta-1, L1 sets are in effect DO loops. The first set being the inner loop and the delta-2, L2 being the outer loop. The "w" in the lower right hand corner is the boundary mode. In this example the boundary mode is "wrap around". Wrap around means that when the number of reads in the displacement direction, the"L1" "L2"'s etc, exceed the number of elements in the row, the reading wraps around to the first element in the row until the "L1" number of data points has been output.

SMART PORT ARCHITECTURE

The Smart Port supplies data to the Arithmetic Unit by executing the addressing sequences specified by MFL's array transformation code. Two Smart Ports read data from memory to the AU, ports C and B, and a third writes the AU output back into memory, port A. Figure 9 shows an implementation of an MFL processor (the Air Force's AOSP).

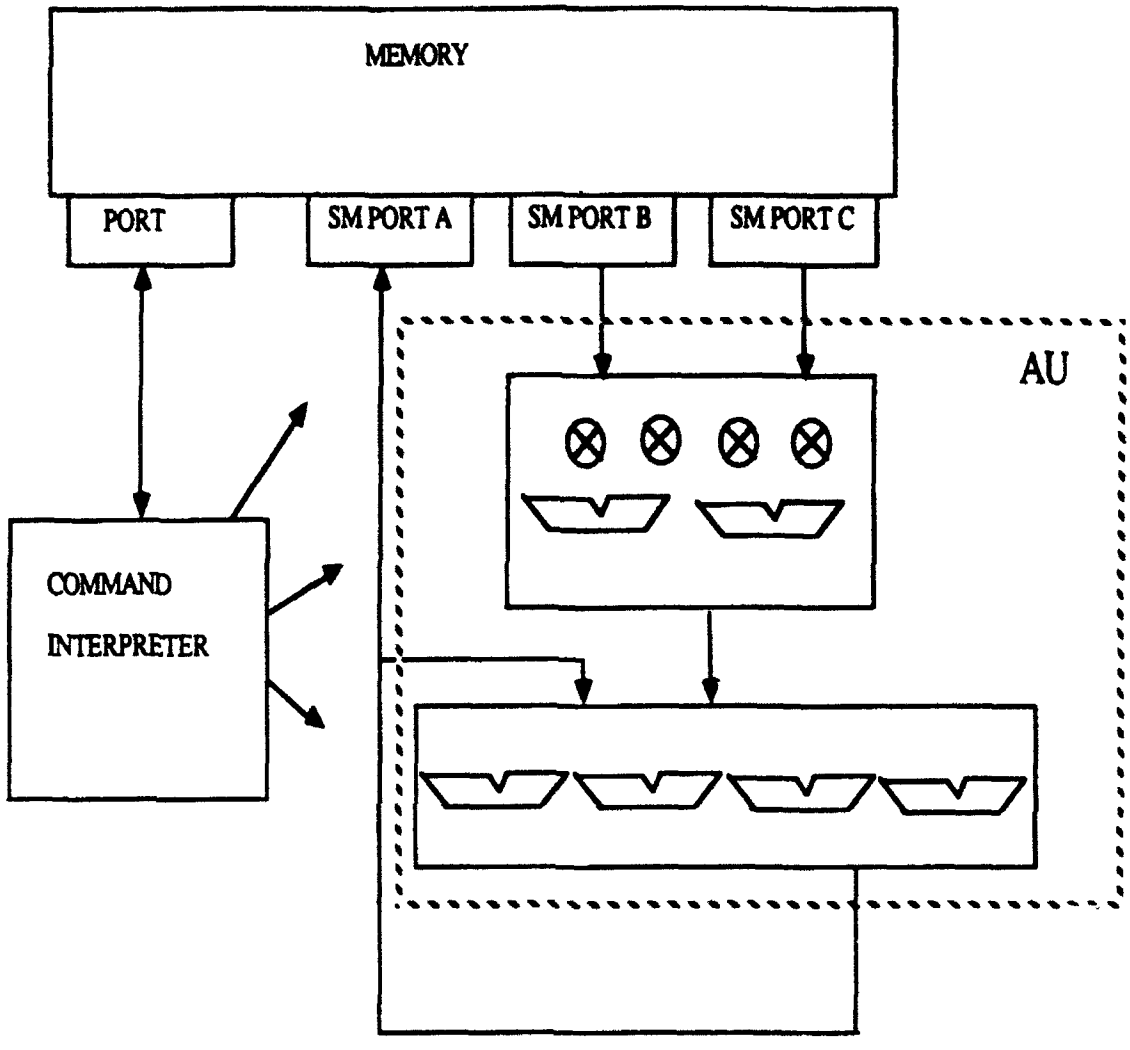


Figure 9 AOSP'S MFSP

Note the AU in the processor of figure 9. It has been optimized, with 4 multipliers and two ALU's in the first stage and 4 ALU's in the second stage, for the class of problems the processor will work on.

The architecture of each AOSP Smart Port is shown in figure 10.

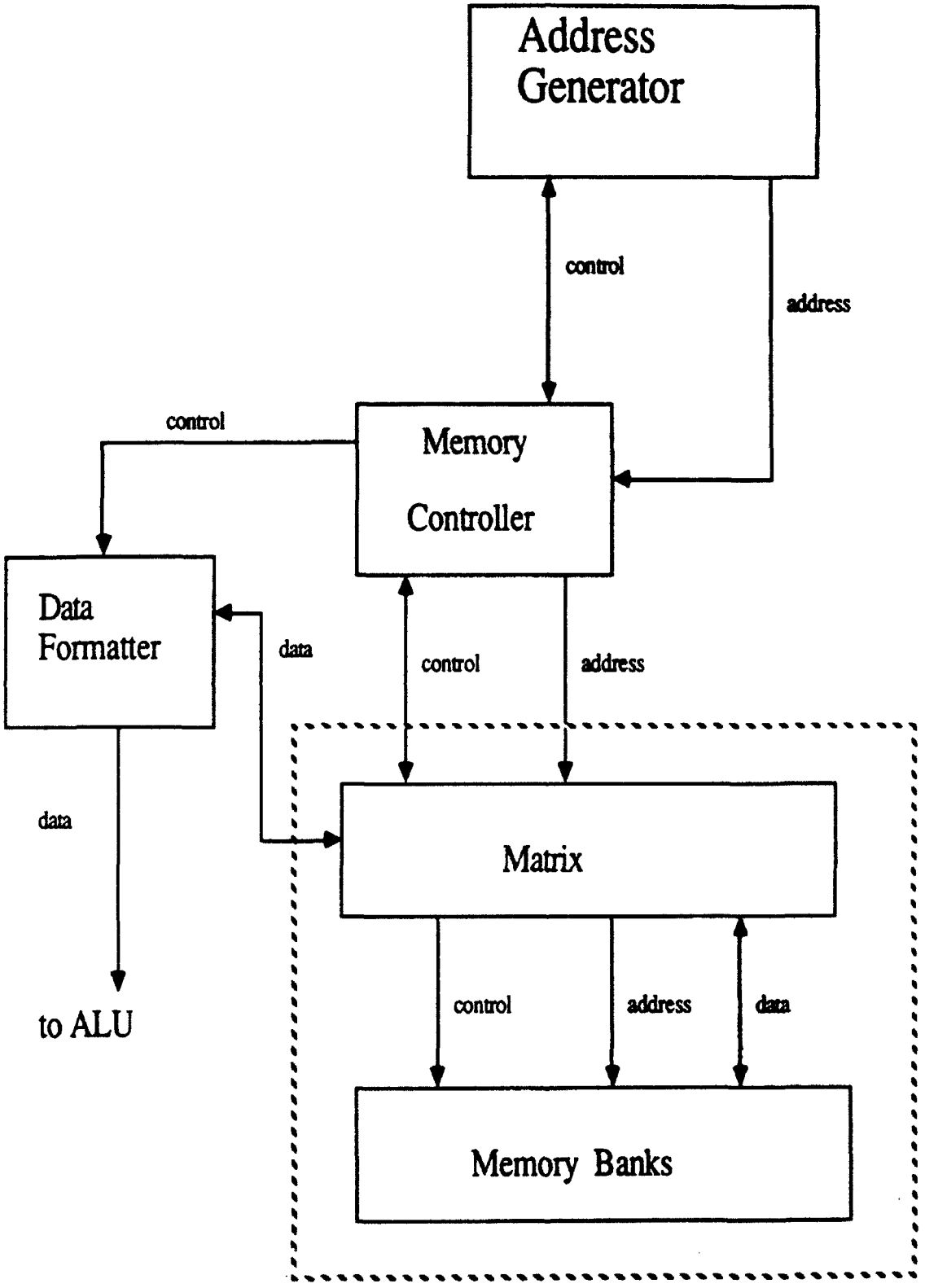


Figure 10 AOSP Smart Port

The AOSP Smart Port includes an address generator, a data formatter, and a memory controller. The address generator produces addresses to access a data array stored in the memory banks. The data formatter translates the data to a form convenient for the AU. In the AOSP configuration, the data is packed in memory in 64 bit words and the data sent to the AU is unpacked and left justified. The memory controller provides control for the address generator and the data formatter, as well as providing interfaces to the AU (control lines) and the CI (ABUS). The memory controller initiates and controls all memory accesses by the Smart Port.

The key piece of hardware in the MFL Smart Port is the address generator and in the next section of the report, the address generator will be covered in detail. The array transformation code of MFL is the program code of the address generator.

ADDRESS GENERATOR ARCHITECTURE

This section will examine the Smart Ports address generator architecture with a few examples of a register by register mapping of MFL code into the address generator registers. These examples will show that, instead of microcode, the Smart Port is programmed by simply passing parameters to it's registers. This concept of parameter passing vs. microcode is a key feature in the flexibility and programability of an MFL design. Thus the array transformation,

the MFL code that drives the Smart Port, sets up the registers of the Smart Port and the Smart Port is simply a processor that has been hardwired to perform one function--the array transformation.

MFL array transformation code has the form:

```
        DATANAME
        C16   10;20
{ ▲4  ▲3  ▲2  ▲1 | ▲0 }
{ L4  L3  L2  L1 | B  }
```

Because the MFL processor is built around the MFL language the processor can be thought of as a high level machine, and the low level program code of the machine is MFL, MFL is then to the user a high level language. The translation from the MFL code to the actual machine code that drives the hardware is then a simple process and the MFL code can be thought of as the microcode of the MFL processor, (the lowest level, simplest mnemonic, closest to the hardware, code of the processor).

In this first array transformation example, the simple, low level relationship of MFL code to architecture can be seen. The MFL code has a one to one relationship with the register contents and programming of the different parts of the hardware. Figure 11 is an overall look at the address generator architecture. On the left in the length section, the length register holds the L1, L2, etc. values and the length counter is a simple counter that increments at each cycle. The comparison of the length counter with the length register determines which loop the address generator is processing. The

examples will demonstrate this process in detail. The registers on the right hold the delta displacements and the delta accumulators hold the current output index. The current output index is then used to compute the address of data to be fetched from or deposited into memory.

The length counter and delta accumulator's registers are like odometers that count up to their respective limits--the L#'s . and then reset. The level number is a 2-bit address that holds the current "level" of the pointers. The boundary adjust checks to see if the current output index is over the actual row and column length and resets the index accordingly, (wrap around and zero fill). The final address is calculated by the formula:
(column index)(row length) + (row index) + (base address) = memory address.

The row and column length adjust is the simple test:

1) if index <0 (wrap around) add row length
   (zero fill) set zero fill flag
2) if index >= row length
   (wrap around) subtract row length
   (zero fill) set zero fill flag

In wrap around mode when the reading hits the end of a row it wraps around to the begining of the row and in the zero fill mode when the end of the row is reached zeros are outputed till the end of the read.

ADDRESS GENERATOR

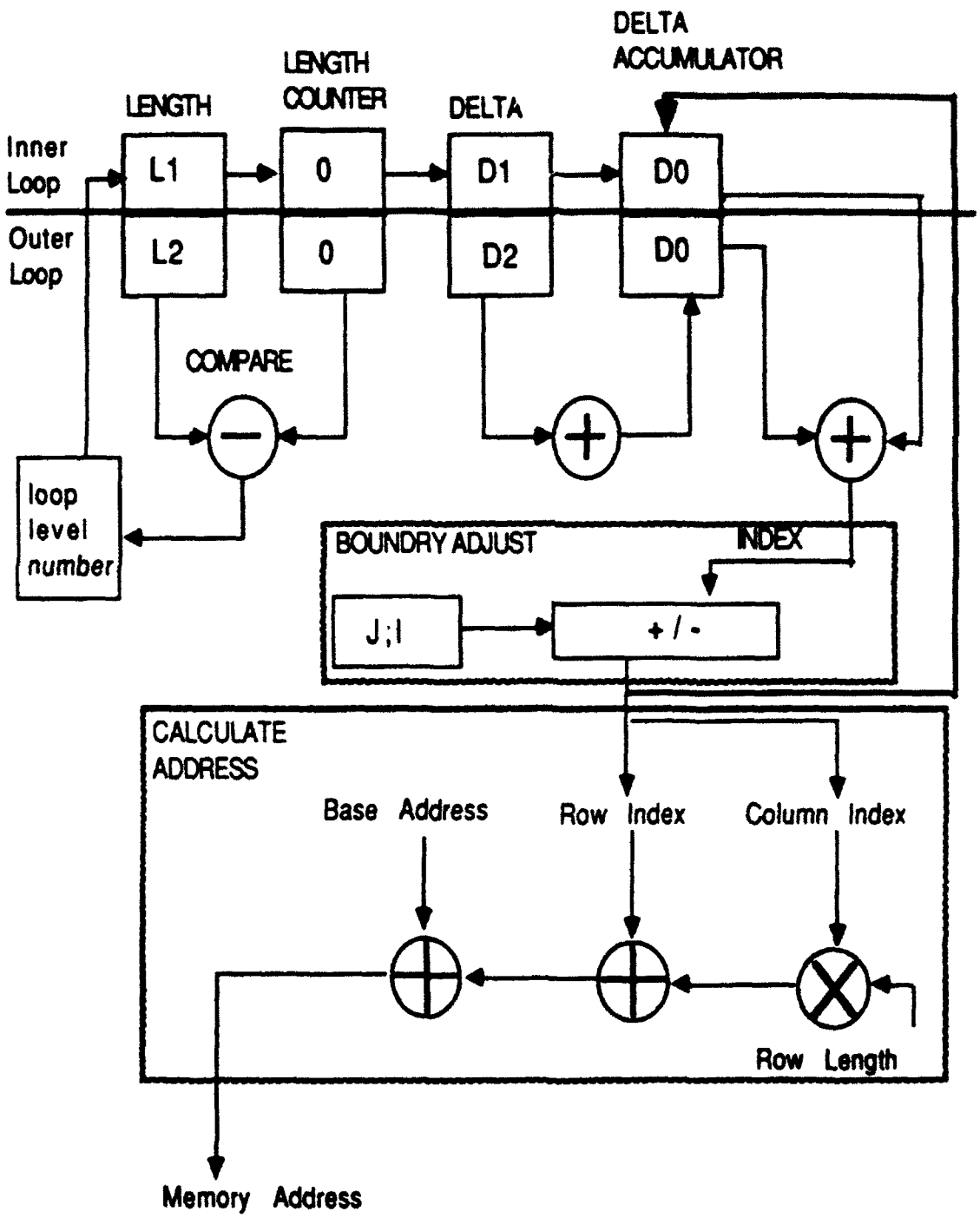
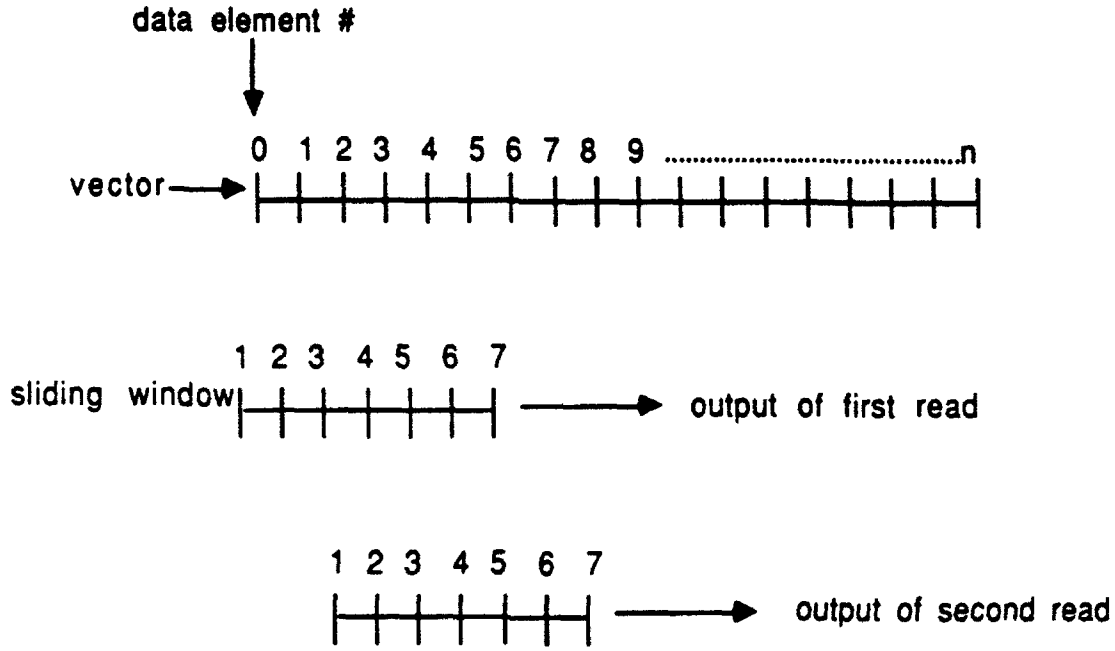


FIGURE 11

The first example is a 7 point sliding window operating across a vector. Figure 11 shows the overall architecture of the Smart Port for this read. The MFL code of this example is: { 2i i | 0 }

{ 25 7 | w }

The code tells the port to read 7 data points, slide two points over, read 7 more, etc, until 25 sets of 7 data points have been read.



Before the Smart Port can execute the required MFL code, the array transformation, the code must first be decoded and loaded into the Smart Port registers. The number of points to be output are

loaded into the address generator's length registers (the inner loop is on top) and the delta displacements are loaded into the delta registers (again the inner loop is on top). Finally, the length counters are initialized to zero and the delta accumulators are initialized to the starting point for the first read. The initial state of the registers is shown in figure 12. Note the one to one mapping of register with the array transformation code.

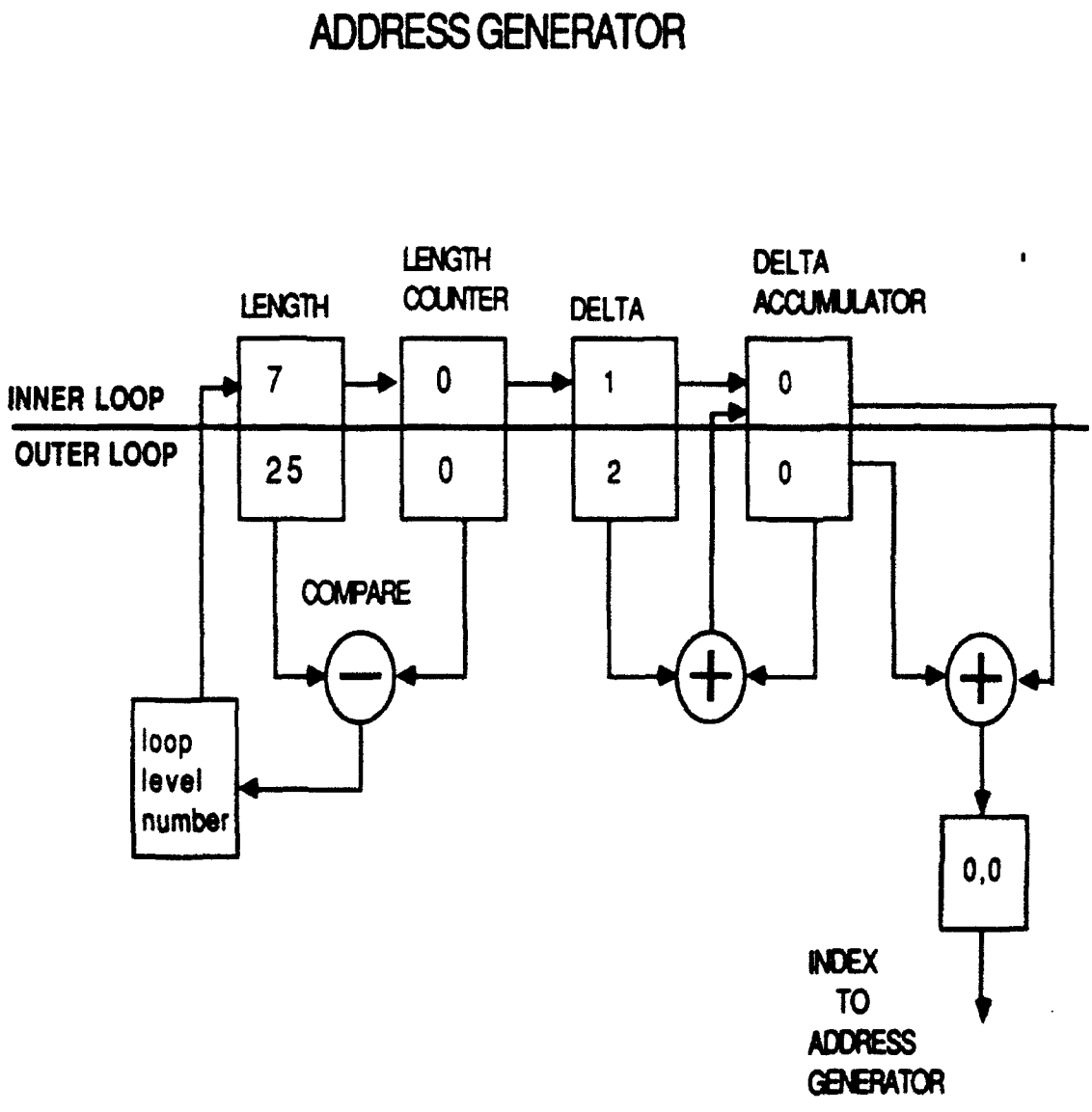


Figure 12

During the first cycle, the delta accumulator values (first index) is output, the length counter is compared with first length register, the counter is less than the length register, the counter is incremented and the contents of the delta register is added to it's accumulator. The results of this first cycle is shown in figure 13.

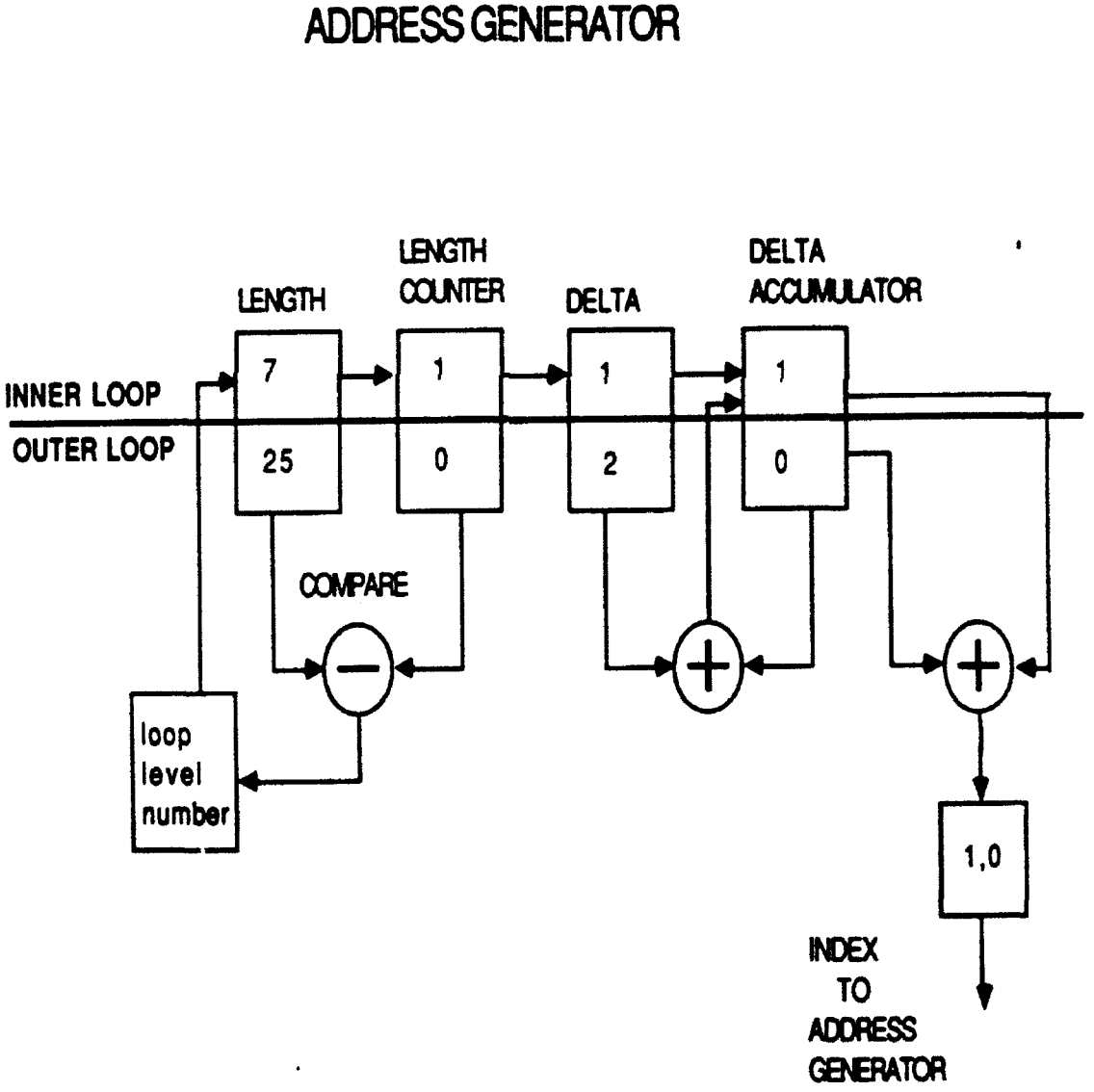


Figure 13

In the next cycle, the current index is first output, the L1 length register is compared with it's counter, the counter is greater than the contents of the length register, so the level stays at one and the length counter is incremented by one. The contents of the delta-1 register are added to its accumulator. This simple incrementing and adding process goes on until the length's counter reaches 7, (7 indexes have been output). The result is shown in figure 14.

ADDRESS GENERATOR

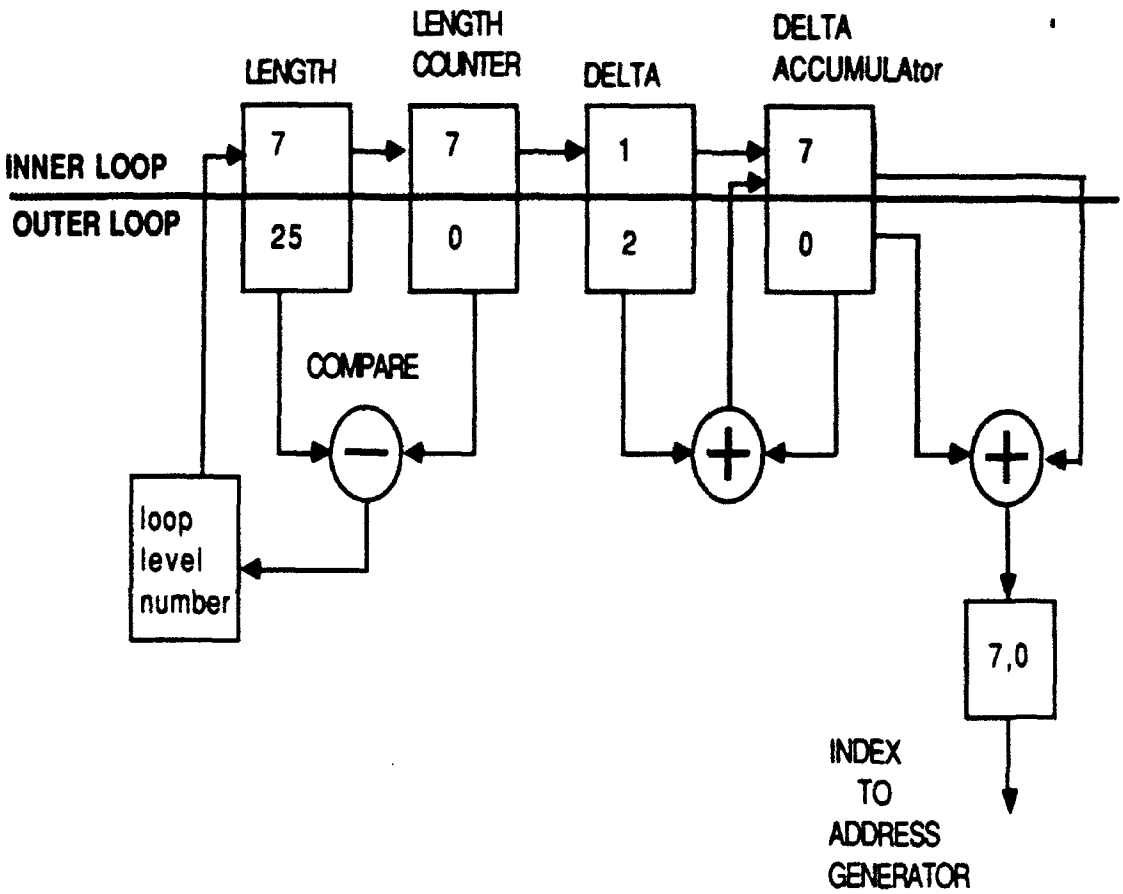


Figure 14

In the next cycle when the length's counter is compared to the L1 register they are equal therefore the level is incremented to level 2, the counter of level 2 is incremented and the accumulator at level 1 is reset. D2 is added to its accumulator and the first level's delta accumulator is reset and the level goes back to one for another round

on the inner loop of the array transformation. The result is shown in figure 15.

This process continues until the outer loop, here L2, has reached it's limit and the array transformation is finished.

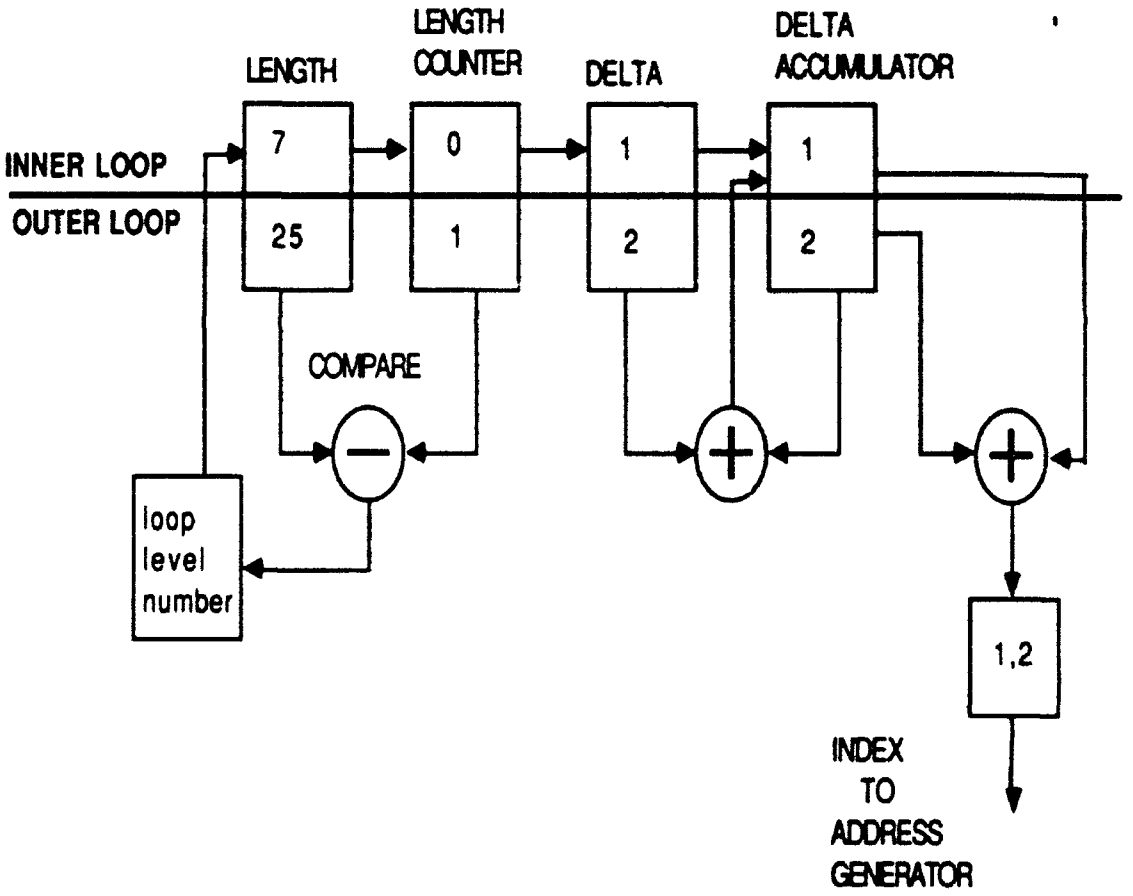


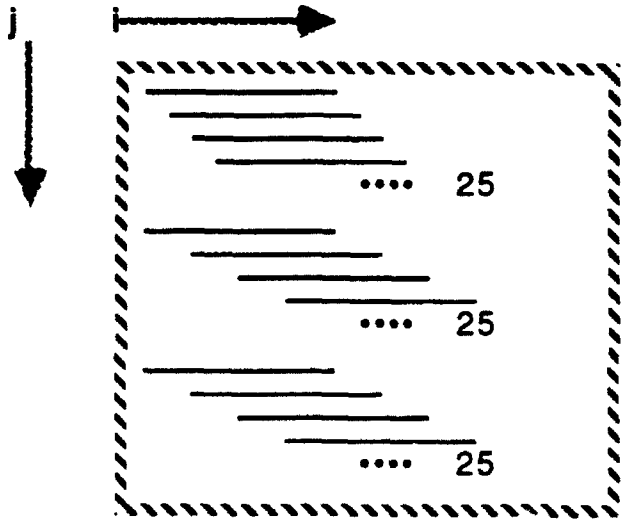
Figure 15

The next example will be a sliding window on a matrix. The output will be a block of data. Consider the MFL code:

{ j 2i i | 0 }

{ 3 25 7 | w }

Graphically the read looks like:



The array being read is a two dimensional array (i X j). The array being output is in three dimensions. Thus each row of the array being read is read as in the first example, forming a 25X7 array, and this read is done three times moving down by one in the column direction. The output passed to the AU is a block of data 3X25X7.

More registers and counters are needed to do this example but the process of loading and cycling through the array transformation is

basically the same. Delta registers are needed for each read in the "i" direction. Here, referring back to the general form of the array transformation, there is a read in the "i" or row direction, in the L1 and L2 slot and in the "j" or column direction in the L3 slot. After the registers are initialized the result is shown in figure 16.

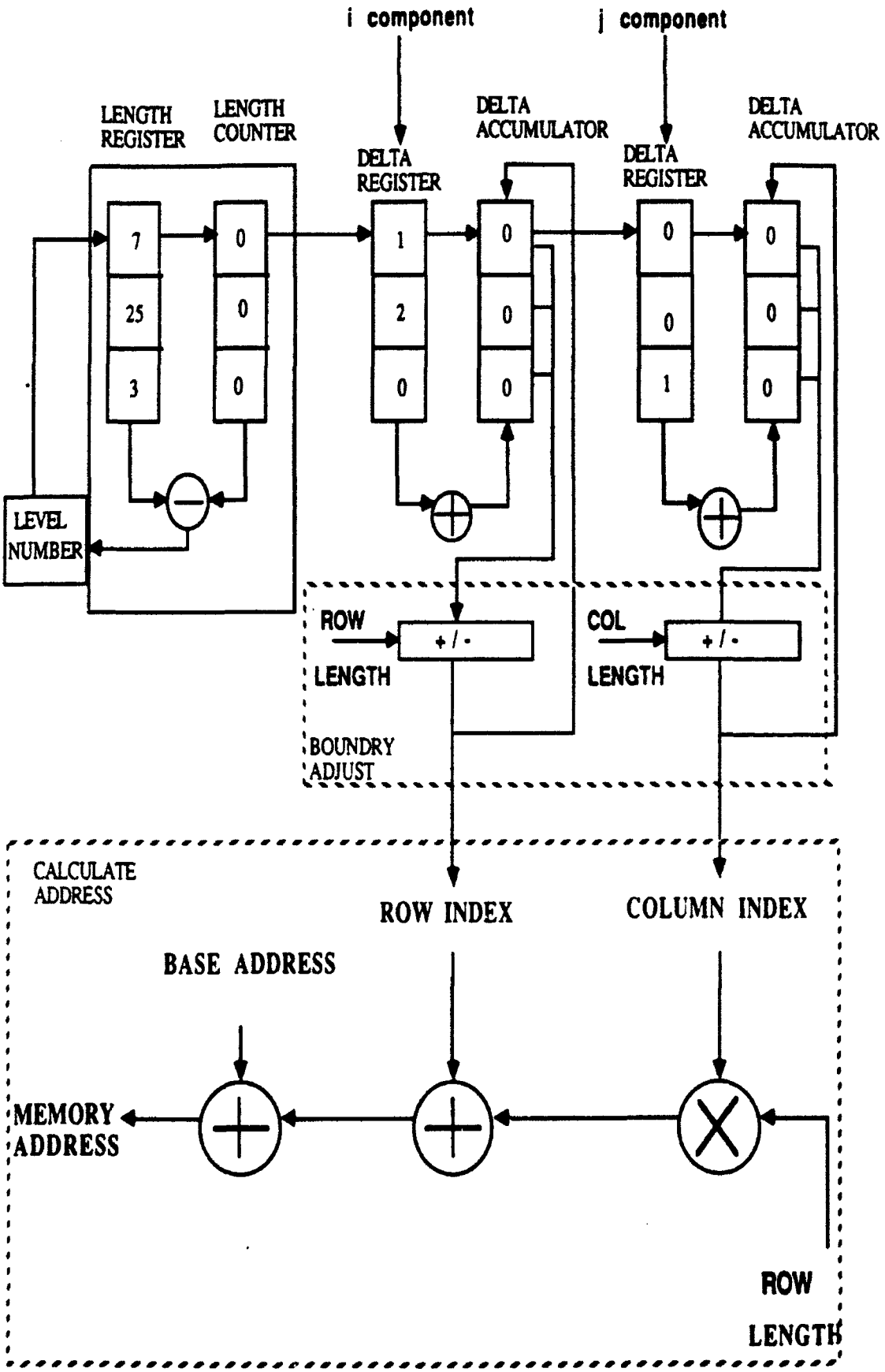

i component
j component
LENGTH REGISTER
LENGTH COUNTER
DELTA REGISTER
DELTA ACCUMULATOR
DELTA REGISTER
DELTA ACCUMULATOR
7
25
3
0
0
0
1
2
0
0
0
0
0
0
1
0
0
0
LEVEL NUMBER
ROW
+/-
LENGTH
COL
+/-
LENGTH
BOUNDRY ADJUST
CALCULATE ADDRESS
ROW INDEX
COLUMN INDEX
BASE ADDRESS
MEMORY ADDRESS
ROW
LENGTH

Figure 16

The next slide shows the result of the first cycle as in the first example: the length counter has been incremented and the delta's have been added to their accumulators. The result is shown in figure 17.

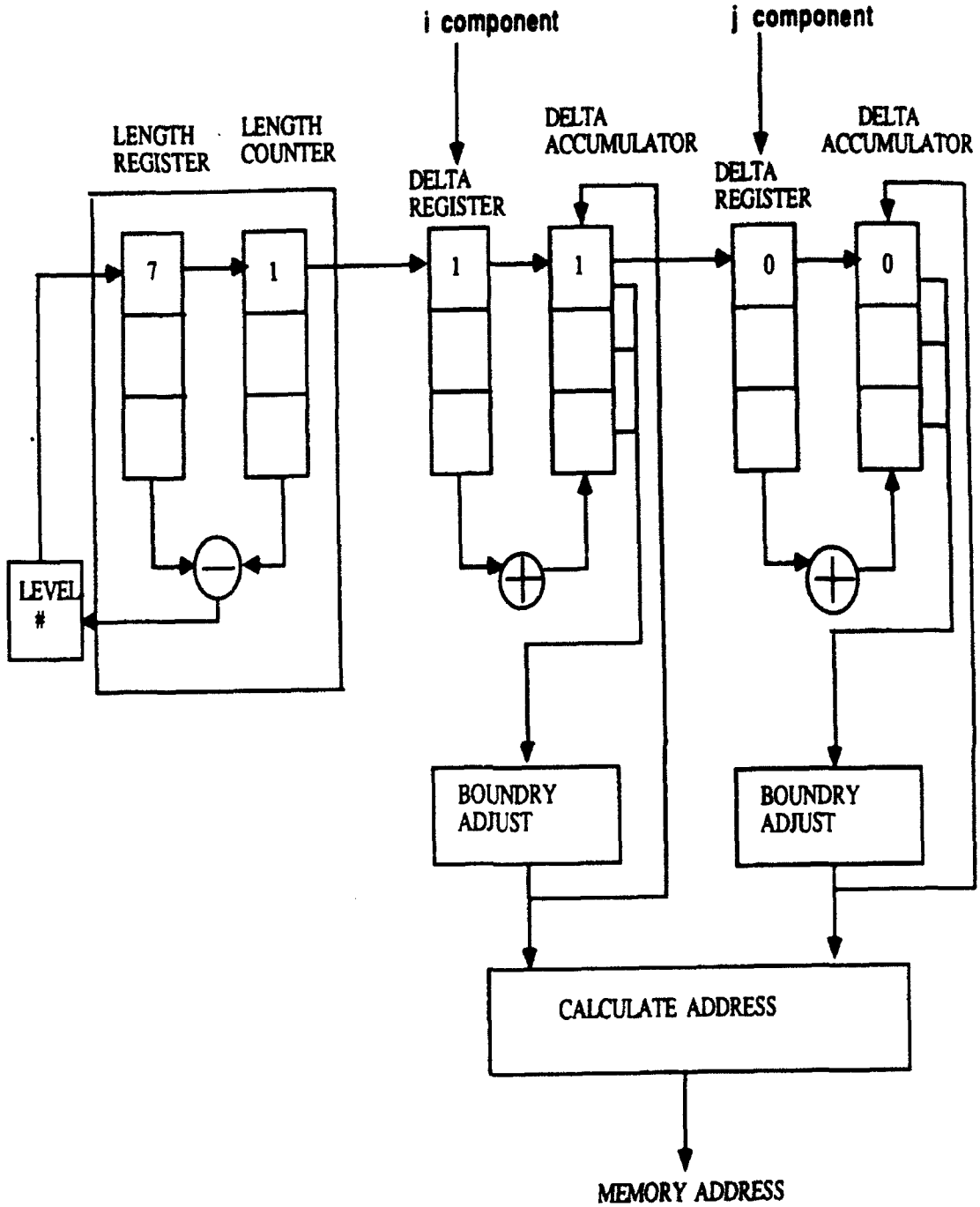


Figure 17

And again after another cycle:

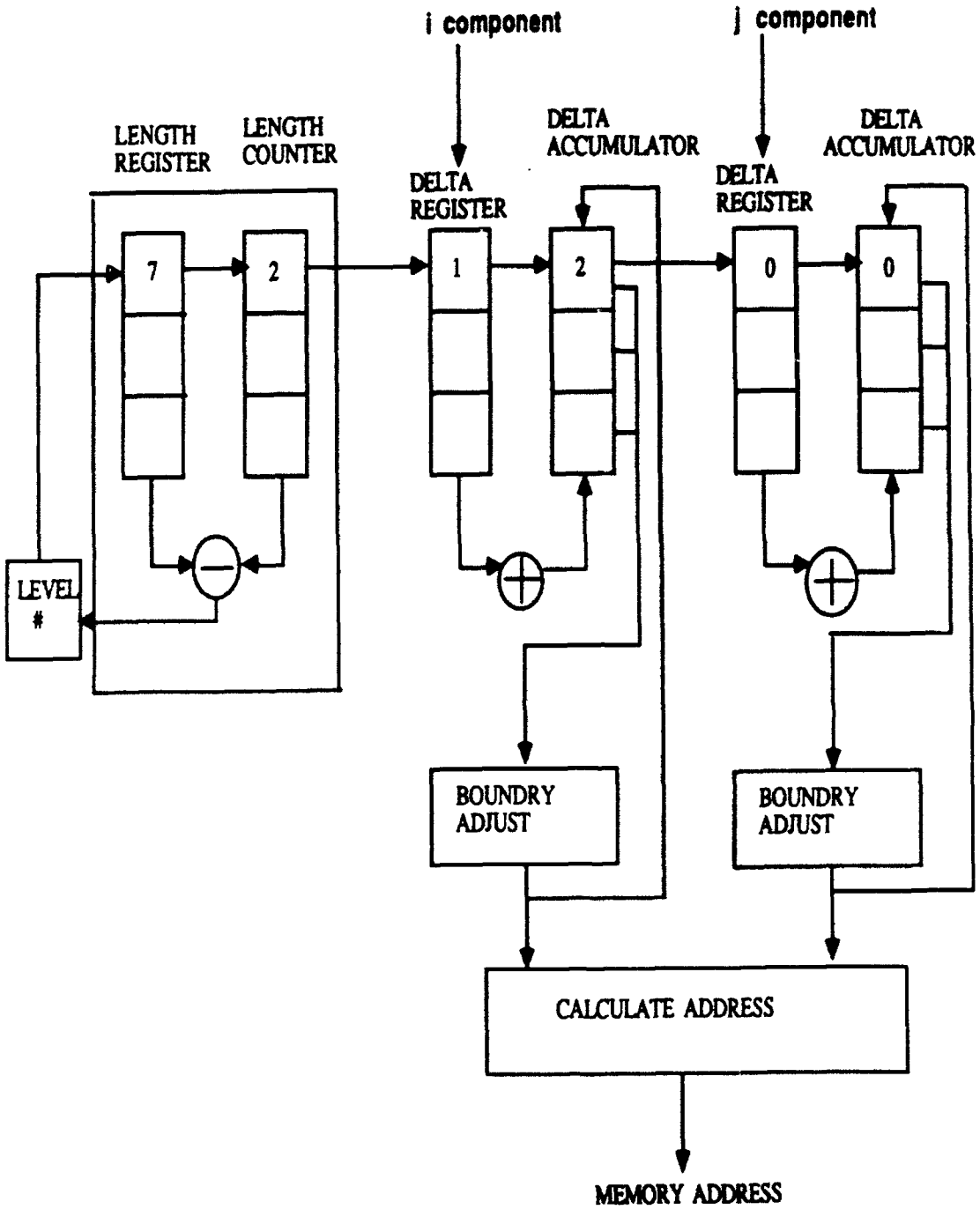

i component
j component
LENGTH REGISTER
LENGTH COUNTER
DELTA REGISTER
DELTA ACCUMULATOR
DELTA REGISTER
DELTA ACCUMULATOR
7
2
1
2
0
0
LEVEL #
BOUNDRY ADJUST
BOUNDRY ADJUST
CALCULATE ADDRESS
MEMORY ADDRESS

Figure 18

The cycles continue at this level until the L1 register equals it's counter register as shown in figure 19:

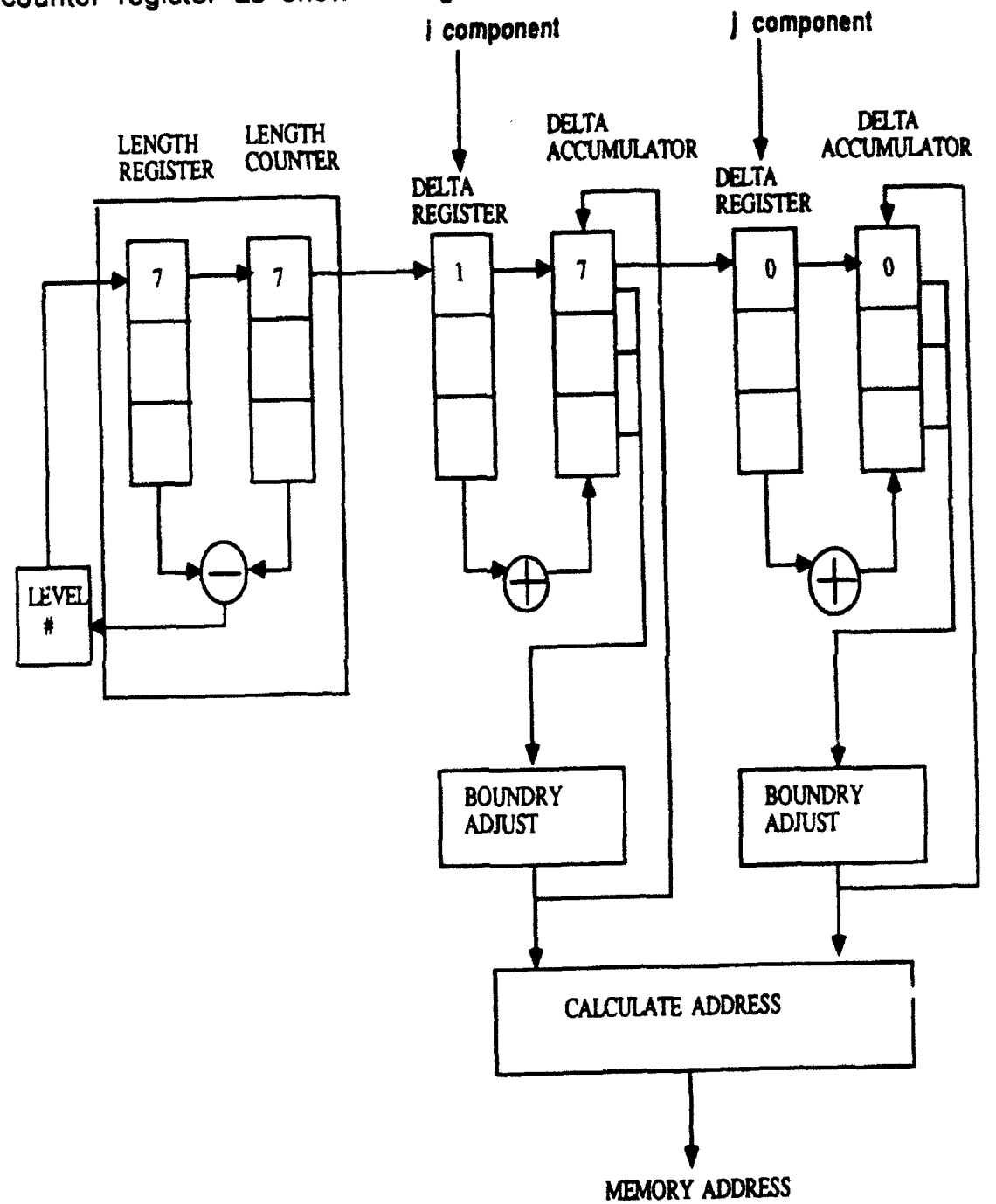


Figure 19

Then the level moves up by one, the second level length counter is incremented and the second level deltas are added to their accumulators

the result is shown in figure 20:

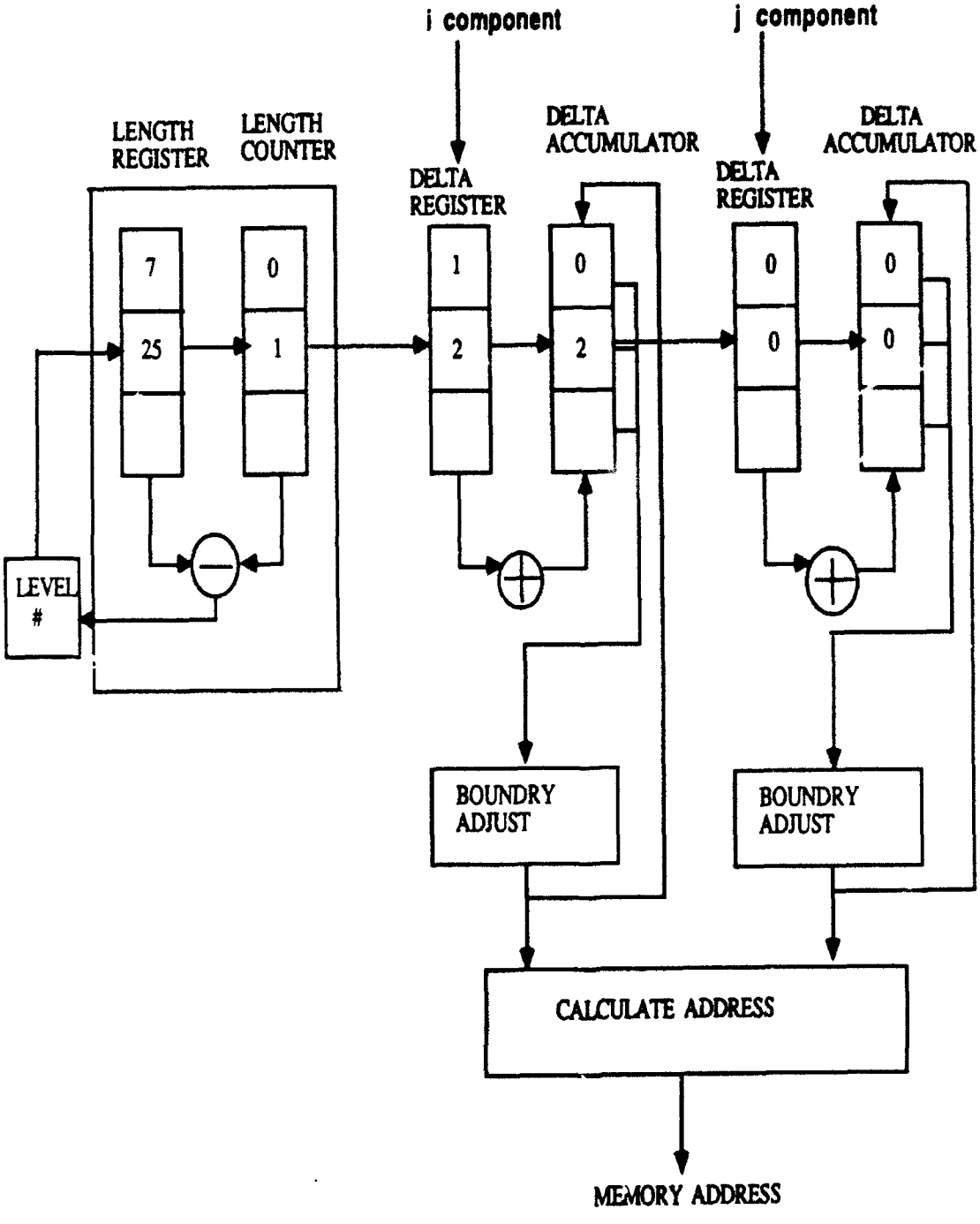


Figure 20

And the level drops back to the first :

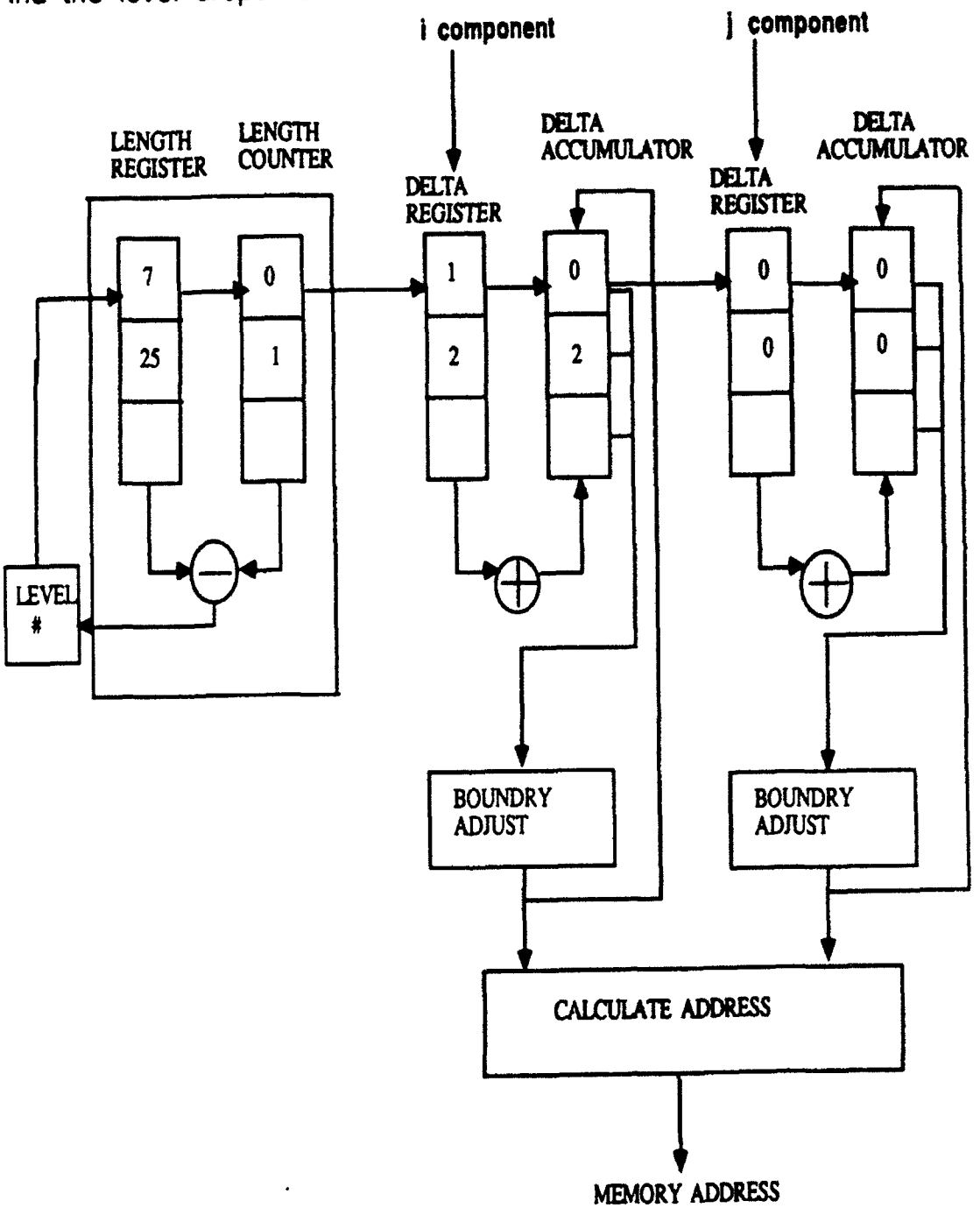


Figure 21

And the inner loop is repeated. This goes on until the L2 counter is equal to the L2 length register as shown in figure 22:

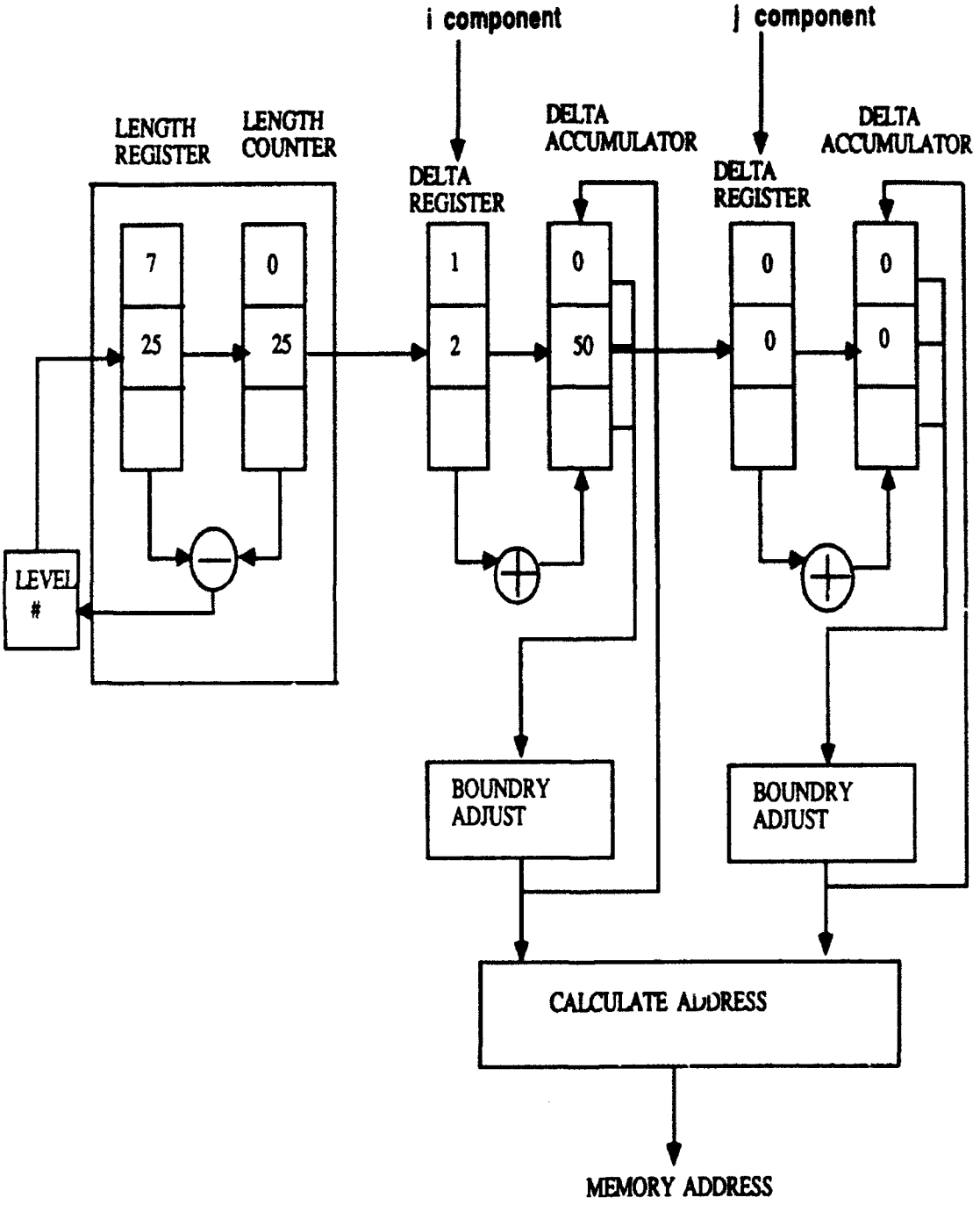


Figure 22

Then the pointer moves up one to the third level, the length counter at that level is incremented and all the length counters below it are cleared. The result is shown in the next figure.

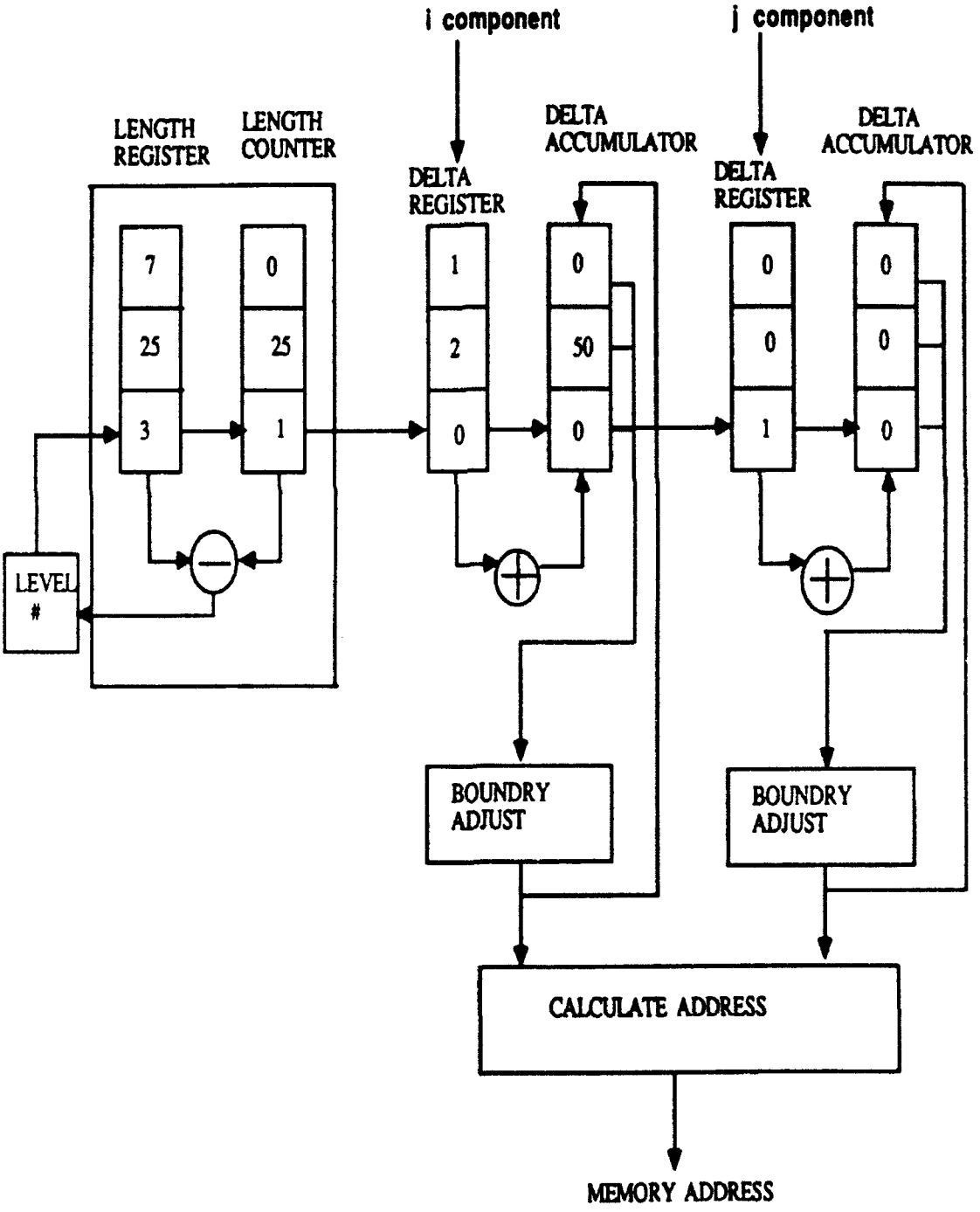


Figure 23

And the level drops back down to the first level, doing the inner loops again until the L3 register equals 3.

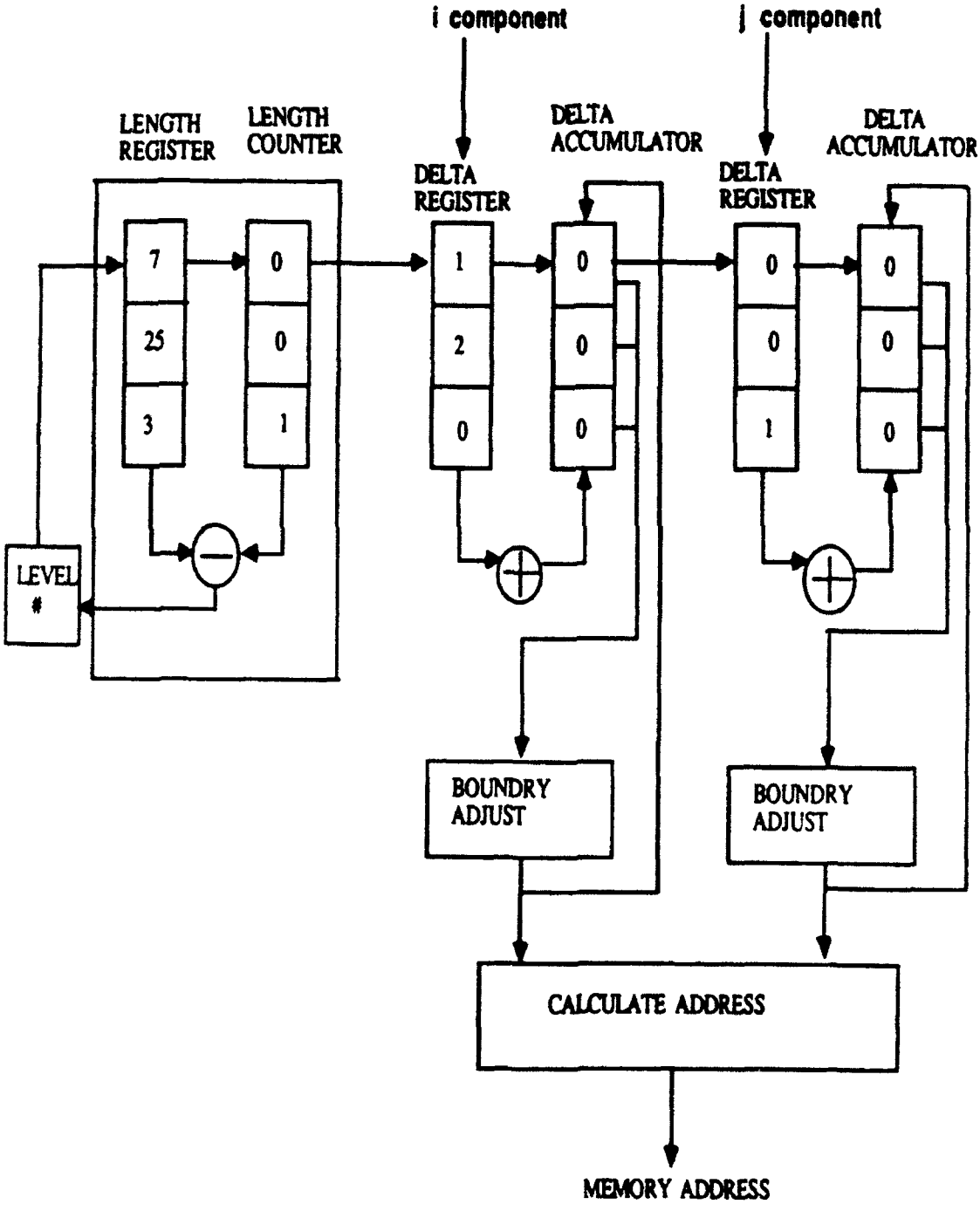


Figure 24

In conclusion smart memory operations can be summarized:

INTELLIGENT MEMORY OPERATIONS

TYPE:	OPERATION:
- Address Generation	- Array Transformation
- Element Selection and alignment	- Unpack/Pack Data
	- Select REAL or IM part
	- Unsigned-Signed Conversion
	- Set Output to Zero/One
	- Left/Right Shift

SPEED ISSUES

A key technology question in the smart port processor is whether or not it can keep up with the AU. AU's generally run at high clock rates and the Smart Port of a MFL processor is now part of the pipe and must feed the AU it's data, formatted and in the right order, in lock step with the AU. A critical technical point to consider is whether or not the Smart Port can keep up with the AU. The AU's pipeline is in effect extended to include the Smart Port. In each clock cycle, the Smart Port(s) must generate 3 addresses, access and deliver 2 data points, and store 1. MFL follows the current trend

of RISC instruction sets. As the memory technologies have increased access times, the RISC trend has been to perform more accesses and in simpler instructions per operation. In the older complex instruction set architectures, because of the slower memory access speeds, the central processor tried to do as much processing as possible on each memory access. Now that memory technologies and therefore memory accesses have caught up to AU speeds, ECL 5ns, GaAs 1ns and static with RAMS 35ns access times, the older complex processing instructions have been broken down to a smaller and simpler instruction sets so that there are more processing cycles and memory accesses per operation and they run at higher speeds. With hardware hardwired to perform complex instructions a speed penalty was paid but now with simpler instructions and faster memories the RISC philosophy has increased processing speeds.

The Smart Port architecture that was reviewed in the previous section shows the amount of processing required for each Smart Port operation. This processing added to the access time of the memory technology used increases the overall access time of the Smart Port. In order to have the fastest access time for the overall Smart Port, a pipelined architecture is suggested. Then the access time of the memory chips themselves become the limiting factor in the speed of the Smart Port and the processing required for each access is done in the pipeline and is transparent to the actual access. An increase in speed beyond the access time of the memory chips would require interleaving between memory banks. In MFL this adds a very diffucult complication because of the array transformations. Array transformations allow the memory to be

read in a completely arbitrary, programable way within a single memory, but in an interleaving strategy reads must be interleaved across memory banks for full throughput. This requires that the data be stored in a prearranged order for the array transformation to work properly. Thus the limiting factor is a straight forward MFL Smart Port design is simply a memory technology with an access time that matches the cycle time of the Arithmetic Unit.

In conclusion the MFL Smart Port is a processor hardwired to perform array transformations. It is programmed by passing parameters to its registers. Because of the amount of processing per cycle to be done in the Smart Port a pipelined processor for the Smart Port is suggested.

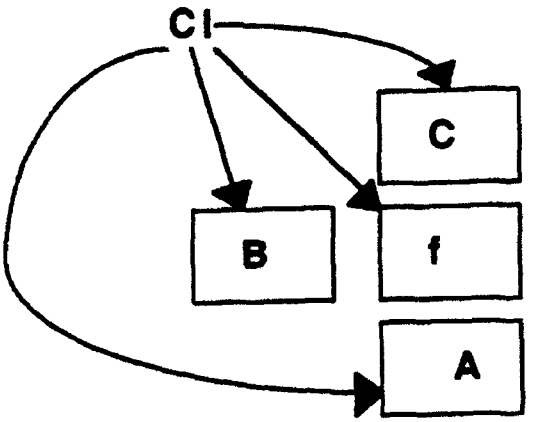
Command Interpreter

The Command Interpreter coordinates the processing of the three Smart Ports and the Arithmetic unit. The role of the Command Interpreter at runtime varies depending on how independent the four MFL processors are: the AU, and the three Smart Ports. The CI is responsible for setting up the Smart Ports and the AU, program sequencing, address modification (looping), and data fetch requests. It obtains the required description of data arguments through their data descriptors and sets up the appropriate pipeline operations required to execute the instructions.

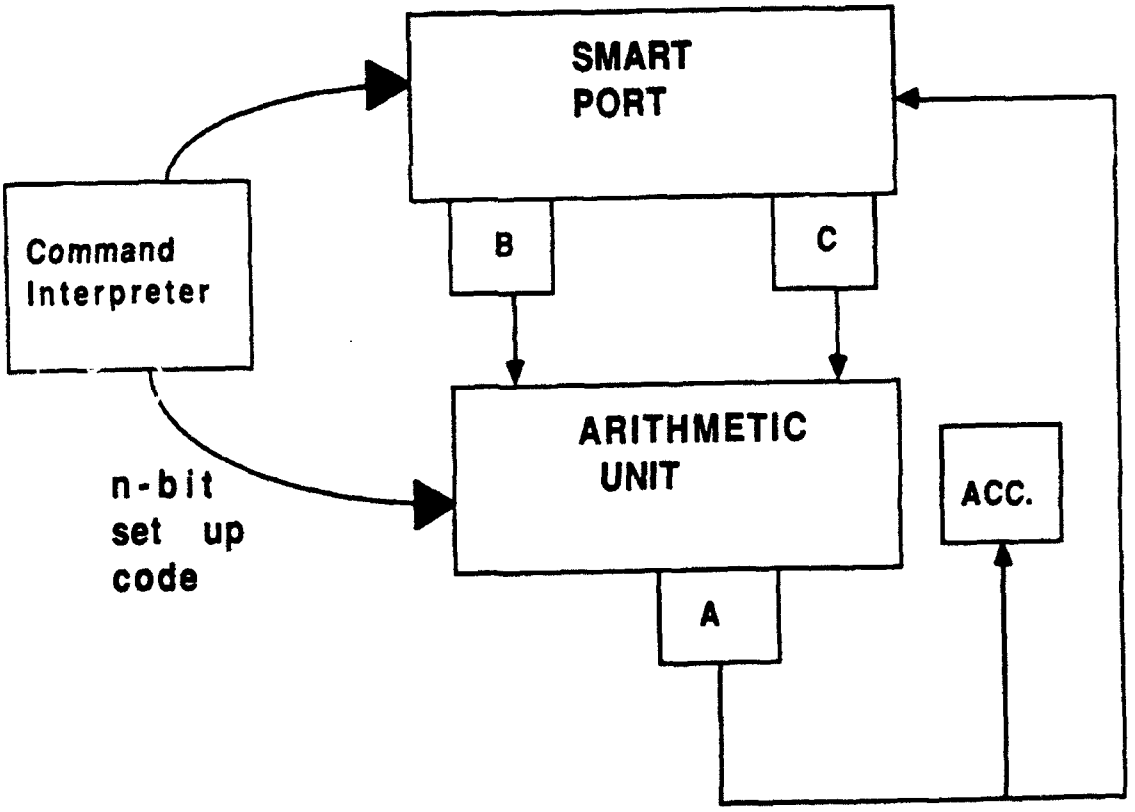
The Command Interpreter function can be performed during run time or at compile time. In the run time mode the four processors of the MFL processor operate in a master-slave function to the Command Interpreter. In general the CI does not involve itself with the detailed control of the AU and Smart Ports after set up in this mode. By releiving the CI of this burden it can anticipate the next instruction and begin the next set up.

The two main functions of the Command Interpreter in this run time mode are:

Interpreting MFL commands from the four code fields,



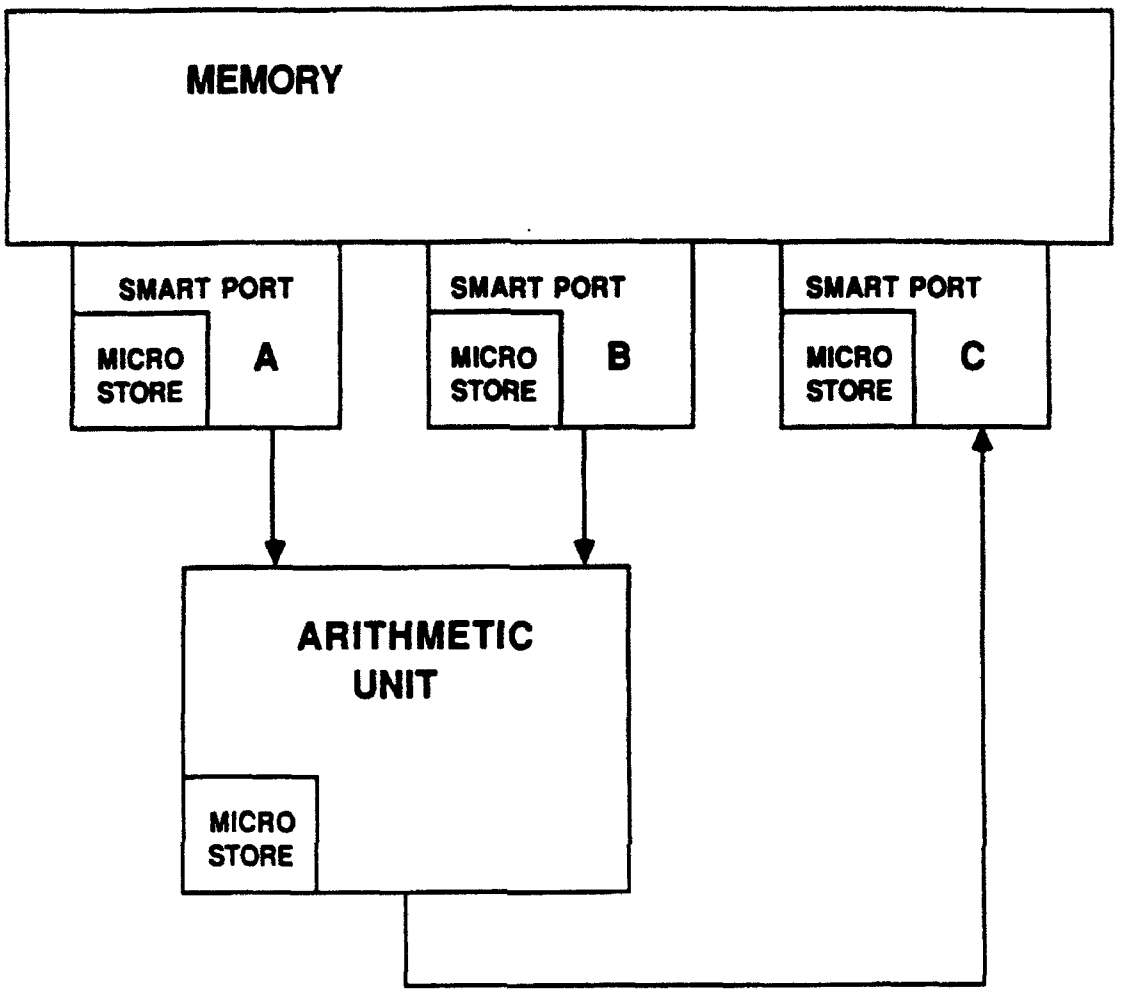
translating these commands into the appropriate machine code and delivering that code to the specified processors-the AU or Smart Ports.



The command interpreter then is a program interpreter and intellegent program counter. It puts the MFL code into hardware and

then is a program sequencer during run time controlling program flow and looping operations.

On the other hand the role of the Command Interpreter during run time could be eliminated by a MFL compiler that handles the Command Interpreter's function at compile time. The sequencing of the four processors would be coordinated and the opcodes and data would be down loaded to the four processors. Then at run time the processor would have no outside involvement. The four MFL processors would then be four completely independent processors running in lock step. This approach helps with one of the potential bottlenecks in a MFL processor: the set up time to pass parameters to the smart ports. The MFL processors in this mode would then each have it's own microstore to hold their run time code as shown in the next figure.



In summary, the Command Interpretter then coordinates the processing of the four independent MFL processors: the Arithmetic Unit and the three Smart Ports, and it can perform this program sequencing at run time or at compile time.

CONCLUSION

In conclusion the MFL processor can be thought of as four independent processors working in lock step. The code in the four MFL code fields drive the different MFL processors. MFL has a low instruction set up overhead because this division of tasks to the four MFL processors: the 3 Smart
Ports and the Arithmetic Unit. MFL also has a mathematically based complete instruction set with a unique array transformation capability.

In this factored parallel operation of a MFL processor each cycle must: generate three addresses(SP)-access 2 and store 1 data point (SP), perform 2 arithmetic operations (AU), and perform the Command Interpretter's functions if the Command Interpretter is operating in the interpretter mode.

MFL has the unique capability of high level to low level code efficiency when the hardware has been built around the language. The break down of this code from a macro level to the machine code is shown in the next figure.

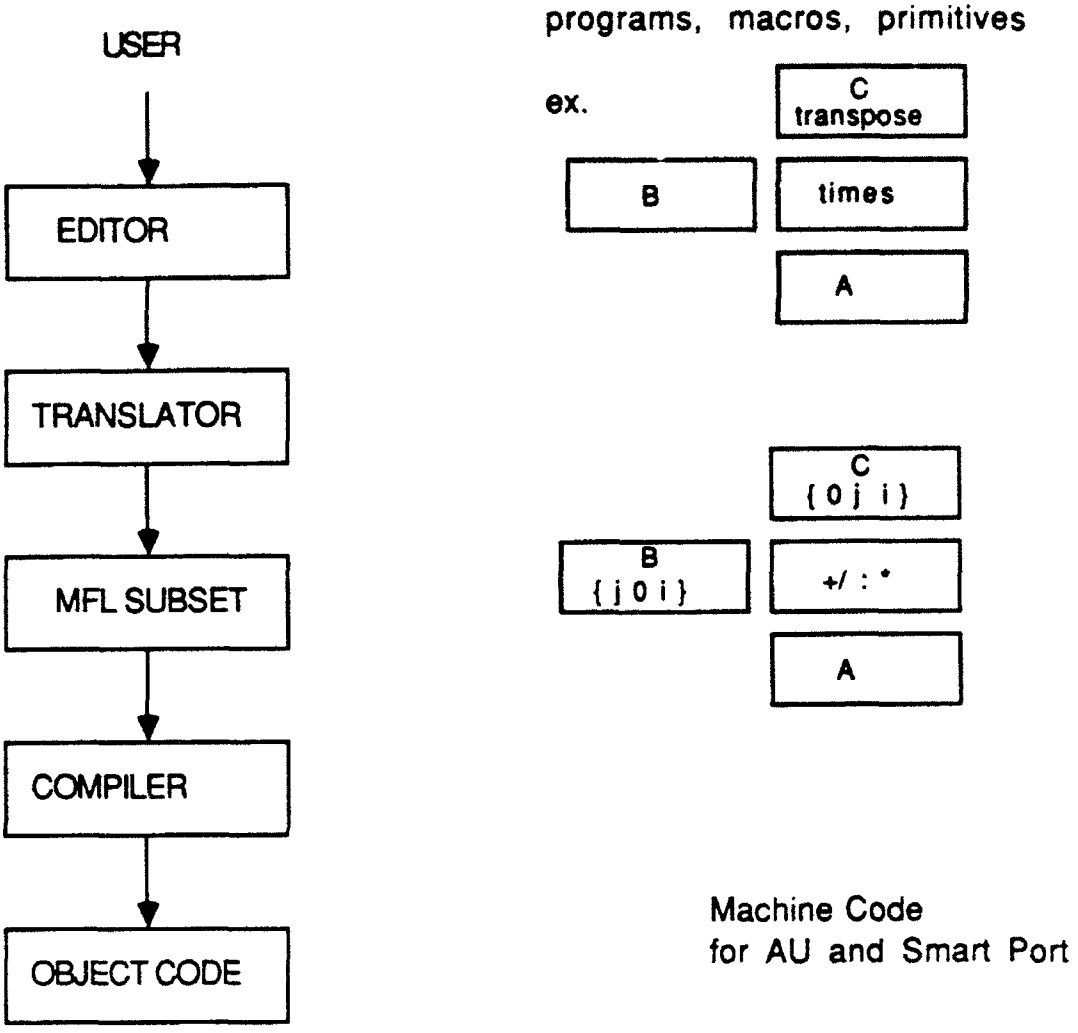


Figure 1

Macros can easily be built from all the lower constructs of the language so that the engineer can program in a very high level of code only invoking commands such as FFT, or BANDSHIFT and supplying the proper inputs. The engineer familiar with the signal processing math can also go below the macro level and write his own customized MFL code. MFL is very different from microcode in this respect. The progression from the highest level to the most primitive, MFL subset

level as shown in figure 1, is structured and orderly and much easier code for a programmer to read than microcode. Microcode is also a very hard language to reread once the programmer has left the program for any length of time, the code quickly becomes unintelligible. MFL, at first glance, is a bit confusing but once the language is learned, the code becomes very readable due to it's close ties to the math of the algorithm and code left for any length of time is easy to pick up and read, if programming with macros, an attribute that most HOL strive for.

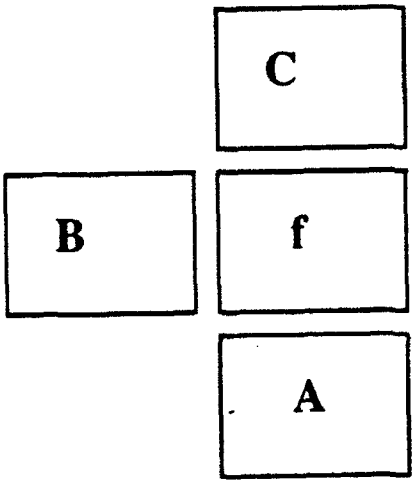
MFL is a complete set of signal processing primitive mathematical and control functions. This set can be trimmed and the hardware then optimized to suit a particular class of problems that the processor will perform. Thus MFL can be thought of as a hardware design philosophy that will deliver a signal processor whose lowest level of code will be programmed in MFL.

The next section of this report is a brief overview of the salient features of the MFL language and the MFL processor.

Synopsis of MFL

A line of Macro Function Language, MFL, code is written in a four field template shown in figure 2.

Graphic Format:



B,C Smart ports READING
A Smart Port WRITING

f Arithmetic Functions

Figure 2

The code within each of these four fields drives a different processor within the MFL processor. In figure 3 the code within the fields marked A, B, C and "f" on the left of the figure, drive the four processors in the generic MFL machine on the right.

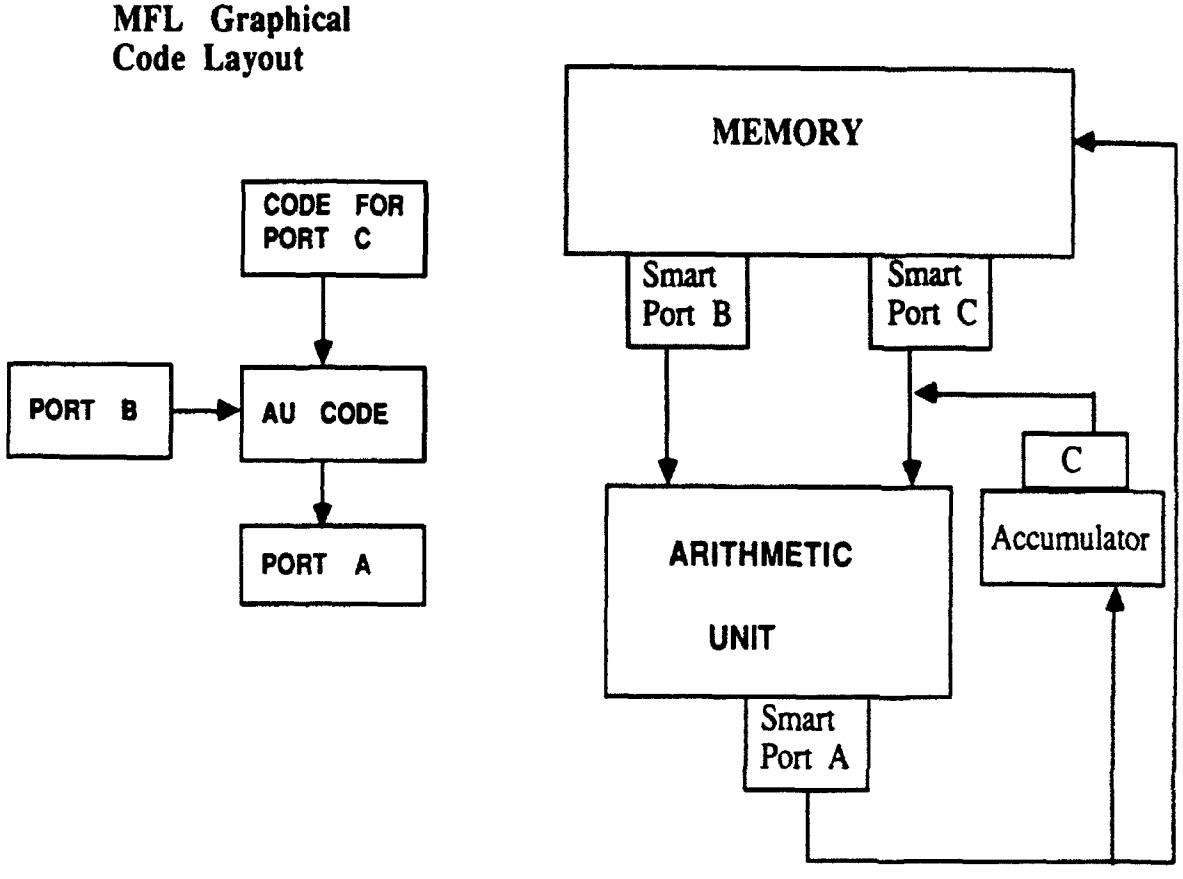


Figure 3

Smart port code, array transformations, has the form:

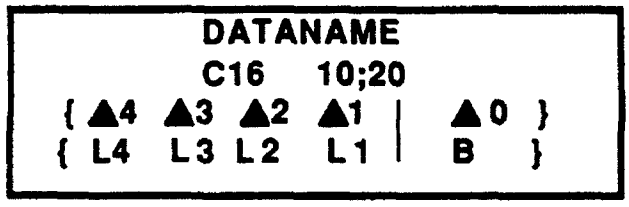


Figure 4

The smart port is a special purpose processor hardwired to perform the array transformation. The smart port is programmed by passing the parameters shown in figure 4 to registers in the smart port, instead of microcoding the smart port. The MFL Smart Port, with the variables from figure 3 for a two dimensional read, are shown with their register's initialized in figure 5.

ADDRESS GENERATOR

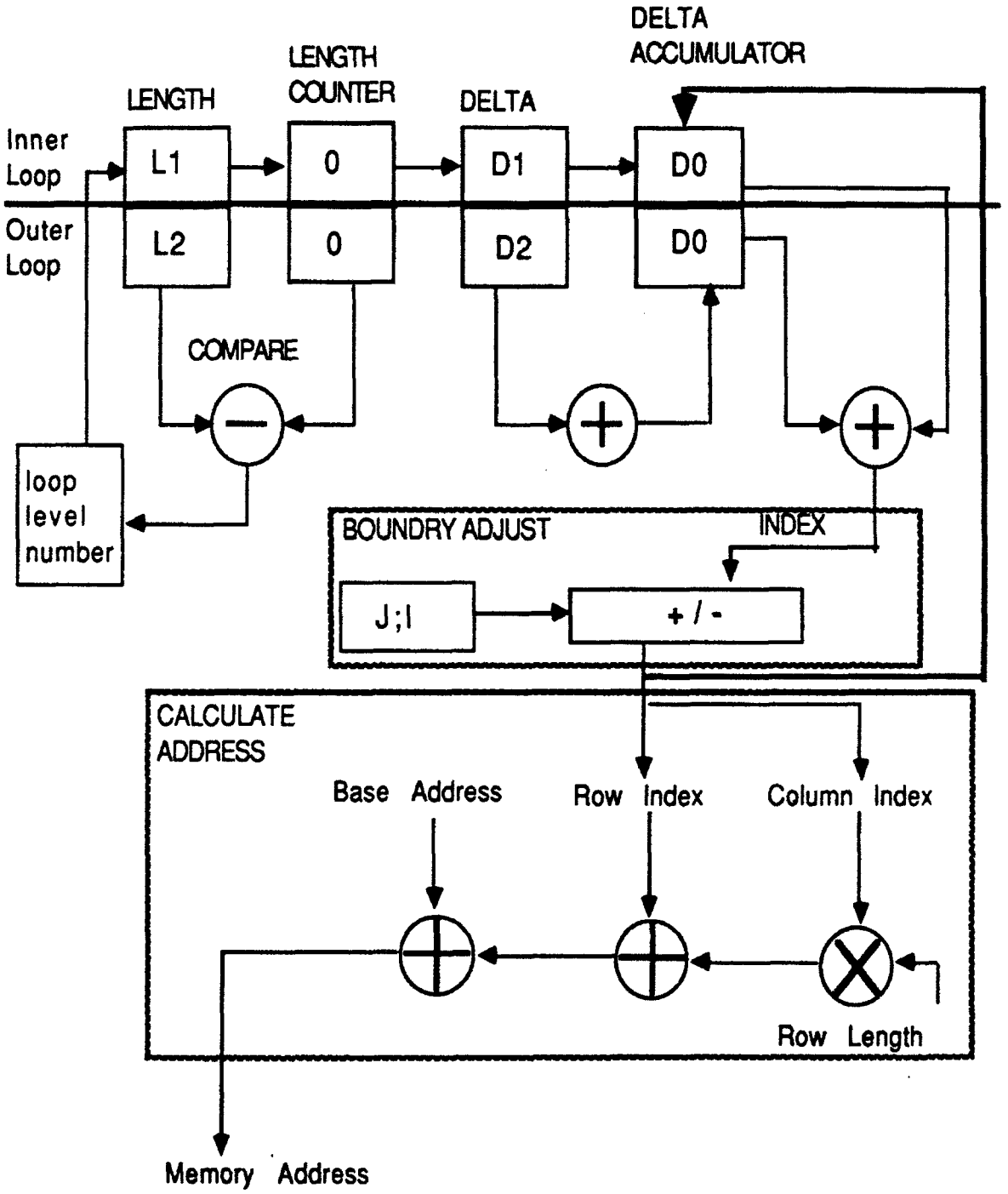


Figure 5

The MFL Arithmetic Unit, AU, is a simplified AU. The data passed to the AU is formatted and in the right order. The AU then is simply the number chruncher of the MFL processor and should be designed to optimize the class of problems that it will work on. The basic primitive MFL AU code is shown in the following figure.

Minimum Function Box Code Primitives

+	Add	∧	And
-	Subtract		
x	Multiply	~∧	Nand
=	Equal To		
*	Conjugate Multiply	∨	Or
<	Less Than		
>	Greator Than	~∨	Nor
	Maximum		
	Minimum	≥	Greator Than or Equal To
		≤	Less Than or Equal To

Minimum Function Box Control Operators

A f B	Single Stream
A B f C	Corresponding Elements
A f / B	Reduction
B f2"f1 C	f2 real part of B & C f1 imaginary part of B & C
'f' C	combine function = (RE C) f (IM C)
f2 : f1	composite - executed right to left

REFERANCES

[1] A. Deerfield, N. Fanala, R. Janssen
"Multisensor Standard Macro Function Study Final Report"
NADC-78188-50, February 25, 1981

[2] K. Brommer, A. Deerfield
"Macro Function Set Formalization"
NADC-85071-50, February 15,1985

[3] A. Deerfield, S. Siu
"Intelligent Memory for use in design of Digital Processing Systems"
NADC-82168-50, September 30, 1983

[4] K. Brommer, A. Deerfield, S. Dymek, S. Scheid, P. Martin
"Intelligent Memory for use in design of Digital Processing Systems-Addendum"
NADC-82168-50, January 30, 1985

[5] K. Brommer, D. Dechert, A. Deerfield, R. Layton
"Macro Function Validation Model Development"
NADC-85103-50, April 30, 1985